O'REILLY®



图灵程序设计丛书

深度学习基础与实践

Deep Learning: A Practitioner's Approach

[美] 乔希·帕特森 亚当·吉布森 著

郑明智 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

译者简介

郑明智

智慧医疗工程师。主要研究方向为医疗领域的自然语言处理及其应用，密切关注大数据、机器学习、深度学习等领域。

数字版权声明



图灵程序设计丛书

深度学习基础与实践

Deep Learning: A Practitioner's Approach

[美] 乔希·帕特森　[美] 亚当·吉布森　著
郑明智　译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

深度学习基础与实践 / (美) 乔希· 帕特森
(Josh Patterson) , (美) 亚当· 吉布森
(Adam Gibson) 著 ; 郑明智 译. -- 北京 : 人民邮电
出版社, 2019.7
（图灵程序设计丛书）
ISBN 978-7-115-51542-1

Ⅰ. ①深… Ⅱ. ①乔… ②亚… ③郑… Ⅲ. ①机器学
习—研究 Ⅳ. ①TP181

中国版本图书馆CIP数据核字(2019)第124039号

内 容 提 要

本书是由两位技术出身的企业管理者编写的深度学习普及书。本书的前四章提供了足够的关于深度学习的理论知识，包括机器学习的基本概念、神经网络基础、从神经网络到深度网络的演化历程，以及主流的深度网络架构，为读者阅读本书剩余内容打下基础。后五章带领读者进行一系列深度学习的实践，包括建立深层网络、高级调优技术、各种数据类型的向量化和在 Spark 上运行深度学习工作流。

本书适合对深度学习的理论和应用感兴趣的开发人员和研究人员阅读。

◆ 著　　[美] 乔希·帕特森 [美] 亚当·吉布森
译　　郑明智
责任编辑　朱　巍
责任印制　周昇亮

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：24.25
字数：573千字　　　　2019年7月第1版
印数：1－3 000册　　　　2019年7月北京第1次印刷
著作权合同登记号　图字：01-2019-3079号

定价：119.00元
读者服务热线：(010)51095183转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京东工商广登字 20170147 号

版权声明

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务还是面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列非凡想法（真希望当初我也想到了）建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

目录

献给我的儿子Ethan、Griffin和Dane，愿你们勇往直前、坚持不懈。

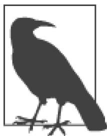
——乔希·帕特森

前言

本书内容

本书前四章着眼于提供足够的理论和知识，为你（实践者）阅读本书剩余内容打下基础。后五章带领你使用 DL4J 进行一系列深度学习的实践：

- 建立深层网络；
- 高级调优技术；
- 各种数据类型的向量化；
- 在 Spark 上运行深度学习工作流。



DL4J 是 Deeplearning4j 的缩写

在本书中，DL4J 和 Deeplearning4j 含义相同，指的都是 Deeplearning4j 库的工具套件。

之所以这样组织本书的内容，是因为我们相信存在这样的需求，即一本既涵盖“足够的理论”，同时又提供足够的实践，使得读者能够构建生产级别的深度学习工作流的书。我们认为这本书双管齐下，很好地满足了这个需求。

第 1 章回顾了机器学习，尤其是深度学习的概念，以期快速地帮助读者为阅读后续章节打下基础。我们设置这一章，是为了让大量的初学者复习或者了解这些概念，希望本书能够帮助到最广泛的群体。

在此基础上，第 2 章介绍了神经网络的基础知识。这一章的主要内容是神经网络的理论，我们会以易于理解的方式讲述这些知识。第 3 章更进一步，带你快速了解深层网络是如何从神经网络演变而来的。第 4 章介绍了 4 个主流的深层网络架构，并且为阅读本书后续内容提供了基础。

第 5 章通过一系列 Java 代码的例子，带领你运用在本书前半部分学到的技术。第 6 章和

第 7 章阐述了普通神经网络调优的基础知识，以及如何对特定的深层网络架构调优。这两章都与平台无关，适合使用任何深度学习库的实践者阅读。第 8 章回顾了向量化技术，还简要介绍了如何使用 DataVec（DL4J 的 ETL 及向量化工作流工具）。第 9 章介绍了如何在 Spark 和 Hadoop 上直接使用 DL4J，展示了 3 个实际的例子，这些例子可以在你的 Spark 集群上运行。

书后提供了多个附录，涵盖了与本书主体内容相关但不甚直接的主题。这些主题包括：

- 人工智能；
- 使用 Maven 管理 DL4J 工程；
- 使用 GPU 进行训练；
- 使用 ND4J API；
- 其他主题。

谁是“实践者”

如今“数据科学”这个词还没有明确的定义，并且经常被用于各种场景。和当今计算机科学中的许多术语一样，数据科学和人工智能（AI）的范围也非常宽广和模糊，这主要是由于机器学习已经扩展到了几乎所有领域。

这种扩展与（20 世纪 90 年代）万维网把 HTML 带到各个行业并把许多新人带入技术领域有相似之处。同样，每天都有各种职业的人——工程师、统计学家、分析师、艺术家——进入机器学习领域。我们编写本书的目标就是为了普及深度学习以及机器学习，让更多人了解它们。

如果你觉得这个主题很有趣，而且正在阅读这个前言，那么你就是实践者，本书就是为你准备的。

目标读者

与那些围绕着小程序的构建来推进的图书不同，本书选择从介绍一系列基础知识开始，带领你完整地走过深度学习的旅程。

我们发现有太多的书忽略了那些企业中的实践者经常需要快速复习的核心主题。基于在机器学习领域的经验，我们决定从入门级别的实践者经常需要温习的内容开始，以便更好地支持他们的深度学习工程。

你可能想跳过第 1 章和第 2 章，直奔深度学习基础知识相关章节。但我们希望你不要跳过，因为它们提供了基本的理论和原理，能帮助你顺利地开始学习更加艰深的深度学习主题。我们为不同背景的读者提供了一些阅读建议。

企业中的机器学习实践者

我们把这一类读者细分为以下两组：

- 在职数据科学家；
- Java 工程师。

在职数据科学家

这类读者通常已经建立过模型，而且非常熟悉数据科学领域。如果你属于这一类读者，那么你很可能会跳过第 1 章，并准备快速地浏览第 2 章。我们建议你从第 3 章开始阅读，因为你很可能已经为学习深层网络的基础知识学习做好了准备。

Java工程师

Java 工程师通常负责将机器学习的代码与生产系统集成。如果你属于这一类读者，我们建议你从第 1 章开始阅读，因为这会帮助你更好地理解数据科学的术语。你应该也会对附录 E 的内容非常感兴趣，因为模型评分代码的集成通常与 ND4J API 直接相关。

企业经营者

我们的一些试读者是财富 500 强企业的经营者，他们对本书内容非常满意，觉得它能帮助自己更好地了解深度学习领域正在发生什么。一位经营者评论说，他大学毕业已有一段时间，而第 1 章对机器学习和深度学习的概念做了很好的回顾。如果你是一位经营者，我们建议你从快速浏览第 1 章开始，这会帮助你回顾一些术语。不过你可能想跳过那些重点讲解 API 和示例代码的章节。

学者

如果你是一名学者，可能想跳过第 1 章和第 2 章，因为在学校已经学过这些内容了。你应该会对神经网络调优以及特定架构调优的相关章节很感兴趣，因为这些内容基于研究，超越了任何具体的深度学习实现。如果你想在 Java 虚拟机（JVM）上进行高效的线性代数计算，那么 ND4J 的相关内容你应该也会感兴趣。

排版约定

本书使用了下列排版约定。

- **黑体字**
  表示名词和重点强调的内容。
- 等宽字体（`constant width`）
  表示程序片段，正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等程序元素，还用于表示模块名和包名、应该由用户直接输入的命令或其他文本，以及命令的输出。
- 等宽斜体（*`constant width italic`*）
  表示应该被替换为用户提供的值或者由上下文确定的值的文本。



该图标表示提示或建议。



该图标表示普通的注记。



该图标表示警告或警示。

使用示例代码

辅助材料（虚拟机、数据、脚本和自定义命令行工具等）可以从 https://github.com/deeplearning4j/oreilly-book-dl4j-examples 下载。

本书的目的在于帮助你完成工作。一般来说，你可以在你的程序和文档中使用本书的代码。只要不是大规模地复制代码，你就不需要联系我们获得授权。举例而言，你写的一个程序用到了本书的几个代码片段，这不需要获得授权。但是，如果你把书中的示例代码刻录到 CD-ROM，并拿去出售和分发，则需要获得授权。在回答问题时引用本书以及本书的示例代码无须获得授权。但如果要在你的产品文档里收录本书中的大量示例代码，则需要获得授权。

欢迎你在使用本书的示例代码时注明出处，但这不是强制要求。通常要注明书名、作者、出版社和 ISBN。例如：*Deep Learning: A Practitioner's Approach* by Josh Patterson and Adam Gibson (O'Reilly). Copyright 2017 Josh Patterson and Adam Gibson, 978-1-4919-1425-0。

如果你认为你对示例代码的使用不在合理使用或上述无须授权的范围之内，那么请通过 permissions@oreilly.com 联系我们。

补充说明

在 Java 代码示例中，我们经常省略导入语句。你可以在实际的代码库中看到完整的导入列表。DL4J、ND4J、DataVec 和更多的 API 信息可在以下网站获得：

> http://deeplearning4j.org/documentation

你可以在以下网址找到所有代码示例：

> https://github.com/deeplearning4j/oreilly-book-dl4j-examples

有关 DL4J 系列工具的更多资源，请查看以下网站：

http://deeplearning4j.org

O’Reilly Safari



Safari（之前称作 Safari Books Online）是一个针对企业、政府、教育者和个人的会员制培训和参考平台。

会员可以访问来自 250 多家出版商的上千种图书、培训视频、学习路径、互动式教程和精选播放列表，这些出版商包括 O’Reilly Media、Harvard Business Review、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Adobe、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 等。

要了解更多信息，可以访问 http://www.oreilly.com/safari。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

对于本书的评论和技术性问题，请发送电子邮件到 bookquestions@oreilly.com。如果你发现任何错误、明显的疏漏或困惑之处，或者有任何改进建议，请给 Josh Patterson 发送电子邮件：jpatterson@floe.tv。

O’Reilly 的每一本书都有专属网页，你可以在那儿找到书的相关信息，包括勘误表[1]、示例代码以及其他信息。本书的网站地址是：http://shop.oreilly.com/product/0636920035343.do

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

注 1：本书中文版的勘误请到 http://www.ituring.com.cn/book/2542 查看和提交。——编者注

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

Josh Patterson 的 Twitter 账号是 @jpatanooga。

致谢

Josh

我邀请了许多比我聪明得多的人来帮助构思和审阅本书内容。像 DL4J 这样规模的项目是无法脱离现实的，在许多社区专家和 Skymind 工程师们的工作的基础上，我才形成了书中的许多想法和准则。

我当时没有想到，与 Adam（在 MLConf 邂逅之后）开发的 DL4J 最终会写成一本书。坦白说，尽管一开始我就参与了 DL4J，但 Adam 做的事情比我多得多。因此，我非常感激 Adam 对这个项目所做的贡献，以及对 JVM 上深度学习想法的贡献，并在前景极不明朗的早期坚持着信念。我想对 Adam 说：是的，你是对的，ND4J 是正确的选择。

写作是一段漫长而孤独的旅程，我要特别感谢 Alex Black 的巨大付出：他不仅审阅了本书，而且还贡献了附录中的内容。Alex 非常了解已出版的神经网络文献，这是编写本书中许多小细节的关键，而且他事无巨细，确保了书中大大小小的知识点都是正确的。没有 Alex Black 的话，第 6 章和第 7 章的内容将至少缩水一半。

Susan Eraly 帮助我们编写了损失函数章节，并且贡献了附录材料。她还订正了书中的许多公式，并且写下了详细的审阅记录。Melanie Warrick 是审阅本书草稿的关键人物，她不但提供了反馈，还为卷积神经网络的内部工作原理编写了注释。

David Kale 是一位勤奋的特设审稿人，他让我在处理许多重要的网络细节和参考文献时小心翼翼。在确定本书的严谨性以及目标读者时，David 提供了学术观点。

每当我就书中内容的取舍发牢骚时，James Long 都是一位严格的听众，并从实践统计学家的角度提出了切实可行的观点。很多时候，对于如何处理复杂的话题并没有明确的正确答案，而 James 是与我从多个角度讨论问题的参谋。David Kale 和 Alex Black 经常提醒我注意数学严谨性，而 James 却总是扮演理性反对者的角色，提醒我们要适度，不要“将读者淹没在数学里”。

Vyacheslav“Raver”Kokorin 提升了自然语言处理（NLP）和 Word2Vec 示例代码的质量。

我不会忘记 Skymind 首席执行官 Chris Nicholson 给予我们的支持。Chris 一直支持这本书，正是他给予我们充足的时间和资源，本书才得以完成。

我要感谢那些贡献了附录内容的人：Alex Black（反向传播、DataVec）、Vyacheslav“Raver”Kokorin（GPU）、Susan Eraly（GPU）、Ruben Fiszel（强化学习）。本书在不同阶段的其他审阅者还有 Grant Ingersol、Dean Wampler、Robert Chong、Ted Malaska、Ryan Geno、Lars George、Suneel Marthi、Francois Garillot 和 Don Brown。如果你在书中发现任何错误，责任都在我。

我要感谢备受尊敬的编辑 Tim McGovern 的反馈和批注，以及他对一个持续了多年、内容

还增加了三章的项目怀着的巨大耐心。他给了我们发挥的空间去做正确的事，我们对此铭记于心。

我还想感谢其他一些人，正是因为他们对我职业生涯的影响才有了这本书：我的父母（Lewis 和 Connie）、Andy Novobiliski 博士（研究生院）、Mina Sartipi 博士（论文导师）、Billy Harris 博士（教授研究生算法课程）、Joe Dumas 博士（研究生院）、Ritchie Carroll（openPDC 创始人）、Paul Trachian、Christophe Bisciglia 和 Mike Olson（招募我到 Cloudera）、Malcom Ramey（为我提供第一份真正的编程工作）、田纳西大学查塔努加分校，还有 Lupi 比萨店（让我在读研期间免受饥饿之苦）。

最后要感谢我的妻子 Leslie，还有我的儿子 Ethan、Griffin 和 Dane。我经常工作到很晚，有时在度假时也是如此，感谢他们对我的耐心。

Adam

我要感谢 Skymind 团队的辛勤工作，正是有他们帮忙审阅本书内容，我们才能继续安心撰写书稿。我要特别感谢 Chris，他容忍了我一边创业一边写书的疯狂想法。

DL4J 开始于 2013 与 Josh 在 MLConf 的一次邂逅，现在它已经发展成为在全世界使用的项目。DL4J 把我带向了全世界，并真正打开了我的世界，带给我无数新体验。

首先，我要感谢我的合著者 Josh Patterson，他编写了本书的大部分，大部分功劳都应该归他。这些年来，为了使本书面世，他投入了许多夜晚和周末，而我则继续研究代码库，并不断地将内容转化成新的特性。

像 Josh 一样，我们的许多队友和贡献者都尽职尽责地帮我们检查数学上是否有错误，如早期加入的 Alex、Melanie 和 Vyacheslav“Raver”Kokorin，以及后期加入的 Dave。

Tim McGovern 耐心听取了我对 O’Reilly 图书内容的一些疯狂想法，我也很感谢他让我为本书取名。

电子书

扫描如下二维码，即可购买本书电子版。



第1章

机器学习回顾

于细微处见真章。

——尼尔·斯蒂芬森,《雪崩》

1.1 学习的机器

过去十年，人们对机器学习的兴趣呈爆炸式增长。几乎每天都会在计算机科学课程、行业会议以及《华尔街日报》中遇到“机器学习”这个词。在所有关于机器学习的谈论中，人们常常把机器学习能够做到的事情和他们期望它做到的事情混为一谈。实际上，**机器学习**是使用算法从原始数据中提取信息，并用某类**模型**表示信息的一门学问。我们使用模型从那些未建模的数据中推断某些信息。

神经网络是机器学习的一种模型，已经存在至少 50 年了。神经网络的基本单元是**节点**，基本上是仿照哺乳动物大脑的神经元构建的。神经元之间的连接也是仿照生物的大脑构建的，也会随着时间（通过“训练”）进化。接下来的两章会深入探讨这些模型是如何工作的。

在 20 世纪 80 年代中期和 90 年代早期，神经网络在架构上取得了很多重要进展。但神经网络需要大量时间和数据才能取得好的结果，这限制了它的应用场景，磨灭了人们的兴趣。21 世纪初，计算机的计算能力呈指数级增长，这个行业经历了之前从未发生过的计算技术的“寒武纪大爆发”。深度学习作为这个领域中一个强有力的竞争者，在计算能力呈爆炸性增长的十年中脱颖而出，赢得了许多重要的机器学习竞赛。这股热度在 2017 年依然没有消退，如今在机器学习的每个角落都能看到深度学习的身影。

本章将深入探讨深度学习的定义。本书结构精心编排，实践者可以利用本书做以下事情：

- 复习线性代数和机器学习相关的基础知识；

- 复习神经网络的基础知识；
- 学习四大主流深度网络架构；
- 使用书中的示例来尝试实现实用的深度网络变体。

希望你觉得这些内容实用且易于理解。我们先快速概览机器学习的基础知识，之后的章节还会介绍一些核心概念，以便你更好地理解本书的剩余内容。

1.1.1 机器如何学习

在定义机器如何学习之前，首先需要定义“学习”。日常生活中，当说到“学习”时，指的是“通过学习、经验或者接受教育来获得知识”。结合我们的主题，可以把机器学习看作使用算法从数据样本中获取其结构描述的做法。计算机学到一些关于结构的信息，这代表原始数据中的信息。结构描述是所构建模型的另一种说法，包含着从原始数据中提取的信息。我们可以使用这些结构或模型来预测未知数据。结构描述（或模型）可以呈现为多种形式，其中包括：

- 决策树
- 线性回归
- 神经网络的权重

每种类型的模型使用不同的方式将规则应用于已知数据，从而预测未知数据。决策树以树结构的形式创建一组规则，而线性模型创建一组表示输入数据的参数。

神经网络有一个所谓的**参数向量**，用于表示网络中节点之间连接的权重。本章稍后会介绍这类模型的细节。

机器学习与数据挖掘

数据挖掘已经出现几十年了，它像机器学习的许多术语一样，常被曲解或错用。基于本书内容，“数据挖掘”的实践被定义为“从数据中提取信息”。机器学习的不同之处在于，它指的是数据挖掘中用于从原始数据获取结构描述的算法。下面是对数据挖掘做法的简要概括。

- 为了学习概念：
  - 需要原始数据的样本。
- 从数据选取行或实例的样本：
  - 这些样本代表数据中特定的模式。
- 机器从这些数据模式中学习概念：
  - 机器通过算法进行学习。

总的来说，上述过程被视为“数据挖掘”。

在 IBM 和斯坦福大学工作的人工智能领域的先驱 Arthur Samuel 将机器学习定义为：

> 在不直接针对问题编程的情况下，赋予计算机学习能力的一个研究领域。

Samuel 开发了一款可以玩西洋跳棋的软件，它能调整策略，因为它能学到输赢的概率与棋盘上某种棋面之间的关系。这种寻找通往胜利或失败的模式，然后识别并强化成功模式的基本做法，支撑着机器学习和人工智能走到今天。

机器能够通过学习达成其自身目标，对此我们已经憧憬了几十年。这也许主要是受到现代人工智能教父 Stuart Russell 和 Peter Norvig 的著作《人工智能：一种现代的方法》的影响，书中写道：

> 一个缓慢、微小的大脑，无论是生物学上的还是电子学上的，是如何感知、理解、预测和操纵比自己更大、更复杂的世界的？

这句话暗示了学习的概念受到了自然界的过程和算法的启发。图 1-1 以图形化的方式展示了人工智能、机器学习和深度学习之间的关系。

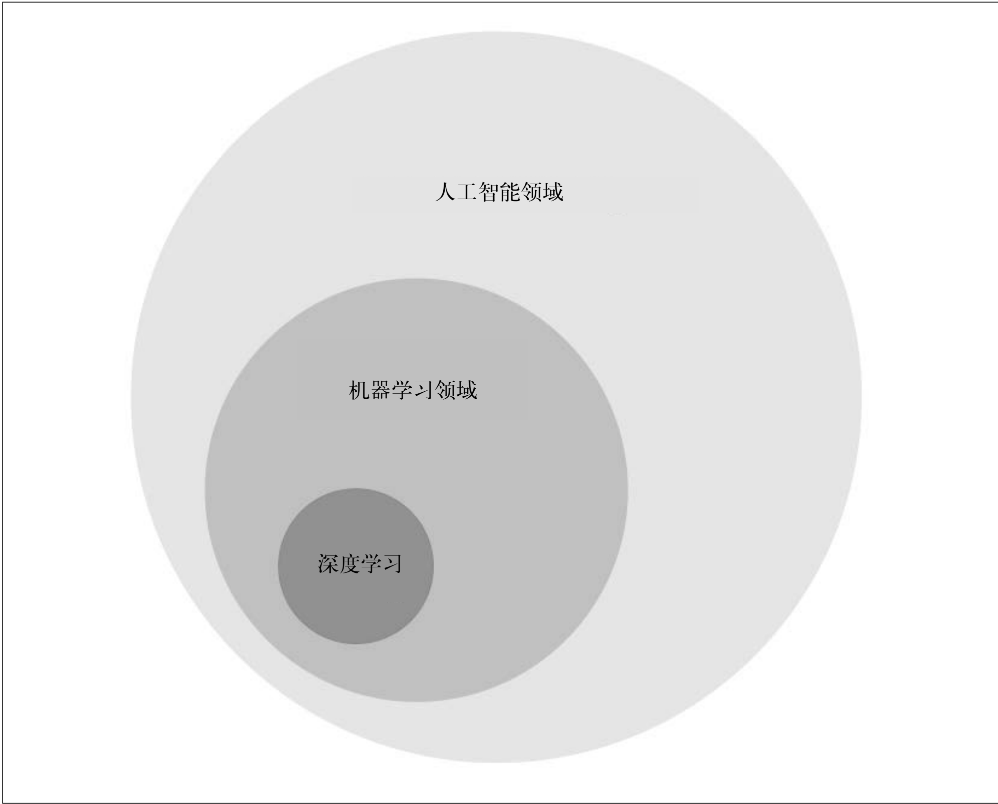


图 1-1：人工智能与深度学习之间的关系

人工智能是个广阔的领域，且已经存在了很长时间。深度学习是机器学习领域的一个子集，是人工智能的子领域。下面快速了解一下深度学习的另一个根源：神经网络是如何受到生物学启发的。

1.1.2　生物学的启发

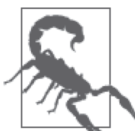
生物学上的神经网络（大脑）大约由 860 亿个神经元组成，每个神经元与许多其他神经元相连。



人脑中全部的连接数

据研究人员保守估计，人脑中神经元之间有超过 500 万亿个连接。即使今天最大的人工神经网络离这个数字也相去甚远。

从信息处理的角度看，生物学神经元是一个可兴奋的单元，可以通过电信号和化学信号处理和传输信息。生物大脑中的神经元被视作大脑、中枢神经系统的脊髓和外周神经系统的神经节的主要成分。本章稍后会介绍，人工神经网络的结构要比它简单得多。



生物神经网络和人工神经网络的比较

生物神经网络要比它的人工神经网络版本复杂几个数量级！

人工神经网络有两个主要特性遵循了大脑工作的一般原理。首先，神经网络最基本的单元是**人工神经元**（或简称为**节点**）。人工神经元模仿大脑的生物神经元，就像生物神经元一样，它们受到输入的刺激。这些人工神经元传递一些（但不是所有）收到的信息给其他的人工神经元，并通常伴有转换。随着本章内容的推进，将会详细讨论这些神经网络中的转换。

其次，通过训练，大脑中的神经元可以只传递那些有助于大脑达成更大目标的信号。同样，我们可以训练神经网络的神经元只传递有用的信号。本章会在这些特性的基础上，介绍人工神经网络如何通过位（bit）和函数来模拟生物神经网络。

生物学对计算机科学的启发

生物学对计算机科学的启发不限于人工神经网络。过去的 50 年中，学术研究还探索了自然界中带给计算机科学灵感的其他主题，例如：

- 蚂蚁
- 白蚁
- 蜜蜂
- 遗传算法

蚁群已经被研究者看作一个强大的去中心化计算机，其中没有一只蚂蚁是会导致整个系统失效的中心节点。蚂蚁不断切换任务，通过定量共识协调等元启发式算法，找到接近最优的负载均衡解决方案。蚁群能够执行清洁、防御、筑巢和觅食任务，同时根据相关需求，为每个任务分配接近最佳数量的工蚁，过程中没有个体蚂蚁直接协调工作。

1.1.3 什么是深度学习

对很多人来说，深度学习很难定义，因为它在过去十年中慢慢地改变了形式。一个有用的定义是，深度学习是处理“两层以上的神经网络”的技术。这个定义的存疑之处是它使深度学习听起来像是自 20 世纪 80 年代以来一直存在一样。我们认为神经网络在最近几年展现出其辉煌成果之前，就已在架构上超越了早期的网络形式（并具有更强大的处理能力）。下面是神经网络发展的一些方面：

- 比之前的网络拥有更多神经元；
- 神经网络中出现了更复杂的层 / 神经元之间的连接方式；
- 用于进行训练的计算能力呈现爆炸式增长；
- 自动特征提取。

基于本书的主题，深度学习被定义为具有大量参数和层的神经网络，拥有以下四种基本网络架构之一：

- 无监督预训练网络（unsupervised pretrained network，UPN）
- 卷积神经网络（convolutional neural network，CNN）
- 循环神经网络（recurrent neural network，RNN）
- 递归神经网络

上述架构还存在一些变体，比如混合了卷积和循环的神经网络。本书会聚焦于上面列出的四种架构的网络。

自动特征提取是深度学习相较于传统机器学习算法的另一大优点。特征提取指由网络决定数据集的哪些特征可以可靠地用于标记数据的过程。历史上，机器学习的实践者花费了他们生命中的数月、数年甚至数十年来人工创建用于数据分类的穷举特征集。在 2006 年深度学习大爆发开始时，最先进的机器学习算法融汇了人类数十年努力积累的用于对输入分类的相关特征。对于几乎所有需要人工微调的数据类型，深度学习在精度上都超过传统算法。这些深度网络有助于数据科学团队将他们的时间、汗水和泪水节省下来，去完成更有意义的任务。

1.1.4 钻进奇幻的兔子洞

深度学习对计算机科学观念的渗透，超过了近代历史上的大多数技术。部分原因是这一技术不仅拥有机器学习模型中顶级的精度，而且它的创造能力甚至让非计算机科学家着迷。一个例子是艺术生成演示，即一个深层网络基于某位著名画家的作品进行训练，然后能够以这位画家的独特风格渲染其他照片，如图 1-2 所示。



图 1-2：Gatys 等人 2015 年的艺术风格图片

这引发了对许多哲学问题的讨论，比如“机器有创造力吗”，以及进一步的“什么是创造力”之类的问题，我们把这些问题留给你后续思考。机器学习已经发展了很多年，它就像季节变化，微妙但稳定，直到有一天你醒来时发现机器已经成为 *Jeopardy* 节目（美国著名知识竞答节目）的冠军，或者击败了一位围棋大师。

机器能变得有智慧，并具有和人类同等的智能吗？人工智能是什么，它能变得多么强大？这些问题尚未得到解答，本书也没有所有的答案。我们只是试图展示机器智能的一些侧面。今天可以通过深度学习的实践来充实我们的生活环境。



关于人工智能的进一步讨论

如果想了解更多关于人工智能的信息，请阅读附录 A。

1.2　提出问题

要想理解有关机器学习应用的基础知识，最佳方式是从提出正确的问题开始。以下是需要定义的事项。

- 我们想要从中提取信息（模型）的输入数据是什么？
- 哪种模型最适合这个数据？
- 基于这个模型，我们希望从新的数据中探索出什么样的答案？

如果能回答这三个问题，我们就可以建立一个机器学习工作流，它将建立模型并产生我们想要的答案。为了更好地完成这个工作流，首先回顾一下为了实践机器学习所需了解的一些核心概念。稍后再看看在机器学习中如何将它们结合起来，然后利用这些信息加深我们对神经网络和深度学习的理解。

1.3 机器学习背后的数学：线性代数

线性代数是机器学习和深度学习的基石，为求解用来建立模型的方程提供了数学基础。



线性代数的一本非常好的入门书是 James E. Gentle 的 *Matrix Algebra: Theory, Computations and Applications in Statistics*。

我们从被称为**标量**的基本概念开始，来了解这个领域的一些核心概念。

1.3.1 标量

在数学中，“标量”一词指的是向量中的元素。标量是用于定义向量空间的实数和字段元素。

在计算中，“标量”与“变量”含义相同，是与符号名配对的存储位置。这个存储位置保存着一个被称为**值**的未知信息量。

1.3.2 向量

基于使用场景，向量的定义如下：

> 对于正整数 n，向量是 n 元组、有序（多）集合或者 n 个数的数组，其中的数被称为元素或标量。

详细说来，即通过一个名为**向量化**的过程创建一个被称为“向量”的数据结构。向量中元素的数量被称为向量的“模”（或“长度”）。向量也可以用来表示 n 维空间中的点。空间意义上，从原点到由向量表示的点的欧几里得距离就是向量的“长度”。

在数学书中，向量经常写成下面的形式：

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

或者

$$\boldsymbol{x} = \left[x_1, x_2, x_3, \cdots, x_n\right]$$

处理向量化有许多不同的方式，可以利用多个预处理步骤，得到具有不同效果的输出模型。本章稍后会详细介绍如何将原始数据转换为向量，然后第 5 章将更全面地讨论。

1.3.3　矩阵

矩阵可以看作一组向量，它们都具有相同的维度（列数）。这样矩阵就是一个二维数组，拥有行和列。

如果一个矩阵被称为 $n \times m$ 矩阵，那说明它有 n 行和 m 列。图 1-3 是一个 3×3 矩阵，用来展示矩阵的维度。矩阵是线性代数和机器学习的核心数据结构之一。

1.0	0.0	0.0
0.0	1.0	0.0
0.0	0.0	1.0

图 1-3：一个 3×3 矩阵

1.3.4　张量

张量本质上是一个多维数组。向量可以看作张量的子集。

矩阵沿 y 轴的行和沿 x 轴的列延伸，每个轴是一个维度，而张量具有额外的维度。张量也有秩，相比较而言，标量的秩为 0，向量的秩为 1。我们也可以看出矩阵的秩为 2。秩为 3 及以上的任何实体都被视作张量。

1.3.5　超平面

另一个需要了解的线性代数对象是**超平面**。在几何中，超平面是比其环绕空间少一维的子空间。在三维空间中，超平面有两个维度。在二维空间中，一维的线是超平面。

超平面是将一个 n 维空间分割成单独“部分”的数学构造，因此在分类等应用中会有用。优化超平面参数是线性建模的一个核心概念，在本章稍后的内容中你将体会到这一点。

1.3.6　相关数学运算

这一节简要回顾你需要知道的常见线性代数运算。

1. 点积

机器学习中一个常见的核心线性代数运算是**点积**。点积有时被称为“标量积”或“内积”。点积计算相同长度的两个向量，并返回一个数。这是通过匹配两个向量中的元素，将它们

相乘，然后将乘积相加得到的。这个计算没有直接涉及复杂的数学理论，但重要的是这个单一的数字包含了大量信息。

首先，点积是对每个向量中每个元素大小的度量。两个具有较大值的向量可以给出较大的结果，两个具有较小值的向量可以给出较小的值。当使用**规范化**方法从数学角度评估这些向量的相对值时，点积就是这些向量的相似度的度量。数学上把两个规范化向量的点积称为**余弦相似度**。

2. 元素积

实践中经常见到的另一个线性代数运算是**元素积**（或哈达马积）。这个运算适用于长度相同的两个向量，它会产生一个相同长度的向量，其中每个元素的值是两个源向量相应元素的乘积。

3. 外积

外积也被称为两个输入向量的"张量积"。取列向量的每个元素，并将其乘以行向量中的所有元素，从而在结果矩阵中创建新的一行。

1.3.7　将数据转换成向量

在机器学习和数据科学的工作过程中，需要分析所有类型的数据。关键的一点是能够处理各种数据类型并将其表示为向量。在机器学习中，我们使用多种类型的数据，例如文本、时间序列、音频、图像和视频。

那么为什么不能简单地把原始数据输入到学习算法中，让它处理一切呢？原因是机器学习基于线性代数，需要求解方程组。这些方程需要浮点数作为输入，所以需要一种将原始数据转换成浮点数据集的方法。稍后介绍方程组求解时会将这些概念联系起来。原始数据的一个例子是经典的鸢尾花数据集：

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

另一个例子是原始文本文档：

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

这两种情况都涉及不同类型的原始数据，但都需要某种程度的向量化，将数据转换为机器学习所需的形式。在某些时候，我们希望输入数据是矩阵形式，但是可以将数据转换为中间形式（如以下代码片段所示的"svmlight"文件格式）。我们希望机器学习算法的输入数据看起来更像序列化的稀疏向量格式 svmlight，如下面的例子所示。

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468 4:0.
  5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723 4:0
  .7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.33333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872 4:0
  .5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107 4:0
  .5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149 4:0
  .5217391304347826
0.0 2:0.7500000000000002
```

这种格式能够被快速地读入矩阵和代表标签（前例中每行的第一个数）的列向量。行内其余被索引的数字也在运行时被作为“特征”插入到矩阵中的适当位置，以便为机器学习处理过程中的各种线性代数运算做好准备。第 8 章将详细讨论向量化的过程。

这里有一个常见的问题：为什么机器学习算法希望数据的形式（通常）为（稀疏）矩阵？为了理解这一点，先快速了解一下求解方程组的基础知识。

1.3.8 方程组求解

在线性代数的世界中，我们对求解如下形式的线性方程组感兴趣：

$$\boldsymbol{Ax} = \boldsymbol{b}$$

其中 $\boldsymbol{A}$ 是由一组输入行向量构成的矩阵，$\boldsymbol{b}$ 是矩阵 $\boldsymbol{A}$ 中每个向量的标签的列向量。拿出前一个例子中前三行序列化的稀疏输出，并将它们的值表示为如下线性代数形式：

```
列 1                 列 2                 列 3                 列 4
0.7500000000000001   0.41666666666666663  0.702127659574468    0.5652173913043479
0.6666666666666666   0.5                  0.9148936170212765   0.6956521739130436
0.45833333333333326  0.3333333333333336   0.8085106382978723   0.7391304347826088
```

这个数字矩阵是方程中的变量 $\boldsymbol{A}$，每个独立的值或每行中的值被看作输入数据的一个特征。

什么是特征

机器学习的特征是输入矩阵 $\boldsymbol{A}$ 中任意列的值，它被用作一个独立变量。特征可以直接从源数据中获取，但大多数情况下，要使用某些转换来将原始输入数据转换为更适合建模的形式。

一个例子是源数据中有四个不同的文本标签的输入列。我们需要扫描所有的输入数据并索引所使用的标签，然后根据每个标签的索引，在 0.0 和 1.0 之间规范化每行中各列的值（0, 1, 2, 3）。这些类型的转换极大地帮助机器学习为建模问题找到更好的解决方案。第 5 章将介绍更多向量转换技术。

我们希望找到给定行中每列的系数，用于给出输出 $\boldsymbol{b}$ 或每行标签的预测器函数。之前用到的序列化稀疏向量的标签如下所示：

```
Labels
1.0
2.0
2.0
```

前面提到的系数成为图 1-4 所示的 $\boldsymbol{x}$ 列向量（也称**参数向量**）。

	训练记录（$\boldsymbol{A}$）					参数向量（$\boldsymbol{x}$）		输出（$\boldsymbol{b}$）
输入记录1	0.7500	0.4166	0.7021	0.5652		?		1.0
输入记录2	0.6666	0.5	0.9148	0.6956	•	?	=	2.0
输入记录3	0.4583	0.3333	0.8085	0.7391		?		2.0

图 1-4：方程 $Ax = b$ 的可视化

如果存在一个参数向量 $\boldsymbol{x}$，使得该方程的解可以直接写成如下形式，则称该方程组“相容”：

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$$

从实际求解的方法的角度来解释 $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b}$ 这个表达式很重要。该表达式仅表示解本身。变量 $\boldsymbol{A}^{-1}$ 是矩阵 $\boldsymbol{A}$ 的逆矩阵，它通过被称为**矩阵求逆**的过程来计算。考虑到不是所有矩阵都可逆，需要一种不涉及矩阵求逆的方法来解方程。其中一种方法被称为**矩阵分解**。一个通过矩阵分解求解线性方程组的例子是利用 LU 分解来求解矩阵 $\boldsymbol{A}$。除了矩阵分解，来看一下求解线性方程组的一般方法。

1. 线性方程组的求解方法

求解线性方程组有两种通用方法。第一种被称为“直接法”，它在算法上有固定的计算量。另一种称为**迭代法**，通过一系列近似和一组终止条件，可以导出参数向量 $\boldsymbol{x}$。当所有训练数据（$\boldsymbol{A}$ 和 $\boldsymbol{b}$）能够在一台计算机的内存中存储时，直接类的方法特别有效。使用直接法求解线性方程组的著名例子是**高斯消元法**和**正规方程法**。

2. 迭代法

当数据无法在一台计算机的主内存中存储时，迭代类的方法尤其有效，并且从磁盘中遍历各条记录使我们能够对更大的数据量建模。现在机器学习中最常见的迭代法的典型例子是**随机梯度下降**（stochastic gradient descent，SDG），本章稍后会讨论。这一领域的其他技术还有**共轭梯度**和**交替最小二乘法**（第 3 章将详细讨论）。迭代法被证明是有效的、可扩展的方法，不仅遍历本地记录，而且将整个数据集分片到机器集群中，周期性地在所有客户端上计算参数向量的平均值，然后在每个本地建模的客户端上更新参数向量（第 9 章将详细介绍）。

3. 迭代法与线性代数

在数学层面，我们希望使用这些算法操作输入数据集。这个限制要求把原始输入数据转换

成输入矩阵 $\boldsymbol{A}$。这里对线性代数的快速回顾告诉我们“为什么”要不辞辛苦地将数据向量化。本书提供了将原始输入数据转换为输入矩阵 $\boldsymbol{A}$ 的代码示例，告诉你“如何”去做。将数据向量化的机制也会影响学习的结果。本书稍后会介绍，在向量化之前，如何在预处理阶段处理数据，以创建更精确的模型。

1.4 机器学习背后的数学：统计学

下面继续本章内容，我们回顾一下必要的统计学知识。我们需要重点掌握统计学中的一些基本概念，例如：

- 概率
- 分布
- 似然

我们还想强调描述性统计和推断统计中一些其他的基本概念。描述性统计包括以下内容：

- 直方图
- 箱形图
- 散点图
- 平均值
- 标准差
- 相关系数

与之相反，推断统计关注如何从样本泛化到总体。下面是推断统计的一些例子：

- p 值
- 置信区间

概率与推断统计之间的关系：

- 概率从总体来推断样本（演绎推理）；
- 推断统计利用样本数据来推断总体特征。

在了解特定样本告诉我们的关于总体的信息之前，我们需要理解与从给定的总体中抽取样本的不确定性。

关于一般统计学，本书不会涉及其他图书已深入介绍的广泛话题。本节目的不在于全面复习统计学知识，而旨在带领你进入相关主题，你可以通过其他资源进行更深入的研究。免责声明结束，下面从定义统计学中的概率开始。

1.4.1 概率

我们将事件 E 的概率定义为一个总是在 0 到 1 之间的数值。在这个背景下，值为 0 意味着事件 E 不会发生，而值为 1 意味着事件 E 肯定会发生。很多时候，这个概率表示为浮点数，但也可以表示为 0% 到 100% 之间的百分数，不会有低于 0% 和大于 100% 的有效概率。例如，概率 0.35 也可以表示为 35%（$0.35 \times 100 == 35\%$）。

测量概率的典型例子是抛一枚质量均匀的硬币，然后观察会出现多少次正面或反面朝上（例如，正反面各 0.5）。样本空间的概率总为 1，因为样本空间代表给定试验的所有可能结果。正如我们可以看到抛硬币的两个结果（“正面朝上”和它的补集“反面朝上”），0.5 + 0.5 = = 1，因为样本空间的总概率必须总是加起来为 1。

事件的概率表示如下：

$$P(E) = 0.5$$

这个表达式读作：

> 事件 E 的概率为 0.5。

概率（probability）与几率（odds）辨析

统计学或机器学习的初学者经常将概率和几率混淆。在继续学习之前，先搞清楚它们的区别。

事件 E 发生的概率定义为：

> $P(E)$ = (E 发生的情况) / (全部情况)

例如，从一副扑克牌（52 张）中抽出 A（4 张）的概率为：

> 4/52 = 0.077

相反，几率被定义为：

> (E 发生的情况) : (E 未发生的情况)

现在扑克牌的例子变成了“抽出 A 的几率”：

> 4：(52 − 4) = 1/12 = 0.0833333⋯

这里，这两个统计学概念的主要区别是分母不同（全部情况与未发生的情况）。

概率是神经网络和深度学习的中心，这归功于它在特征提取和分类这两大深度神经网络功能中扮演的角色。如果你需要更全面地复习统计学，请参阅 Boslaugh 和 Watters 编写的 *Statistics in a Nutshell: A Desktop Quick Reference* 一书。

进一步定义概率：贝叶斯方法与频率方法

统计学中有两个不同的流派，分别称为**贝叶斯主义**和**频率主义**。这两大流派的基本区别在于如何定义概率。

在频率方法看来，概率只在可重复测量的情况下有意义。当测量某物时，收集数据的设备的差异会导致结果有微小的变化。重复测量多次后，给定值的频率就表示测量该值的概率。

而使用贝叶斯方法时，概率的概念扩展到涵盖陈述的确定性方面。概率是我们对测量结果的认知的陈述。对于贝叶斯方法来说，我们自己对事件的认知基本上与概率有关。

对变量估计值进行陈述之前，频率方法依赖许许多多的盲试。而贝叶斯方法处理的是对变量的“信念”（数学术语为“分布”），并在新信息到来时更新对变量的信念。

1.4.2 条件概率

当我们想知道一个给定事件在另一个事件发生的前提下发生的概率时，将它表示为**条件概率**，表达形式如下：

$$P(E|F)$$

其中 E 是我们对其概率感兴趣的事件。

F 是已发生的事件。

下面是表示一个心率正常的人在医院就诊时在 ICU 死亡概率较低的例子：

$$P(\text{ICU死亡} \mid \text{非正常心率}) > P(\text{ICU死亡} \mid \text{正常心率})$$

有时第二个事件 F 被叫作“条件”。在机器学习和深度学习中，条件概率很有用，因为我们经常对何时发生多个事件以及它们如何相互影响感兴趣。使用机器学习构建分类器时，就会用到条件概率：

$$P(E|F)$$

其中 E 是标签，F 是用于预测 E 的事实的一些属性。比如根据每位病人在 ICU 的测量数据（这里的 F）来预测死亡率（这里的 E）。

贝叶斯定理

条件概率一个更常见的应用是贝叶斯定理（或贝叶斯公式）。在医学领域，它被用于计算在一个特定疾病的检查中结果为阳性的患者实际上患有该疾病的概率。

对于任意两个事件 A 和 B，贝叶斯公式定义如下：

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

1.4.3 后验概率

在贝叶斯统计中，考虑证据之后分配给随机事件的条件概率称为随机事件的后验概率。我们将从实验中收集的证据作为随机变量，将后验概率分布定义为依赖这个证据的未知量的概率分布。我们会在 softmax 激活函数（本章稍后解释）中见到这个概念的作用，其中原始输入值被转换为后验概率。

1.4.4　分布

概率分布是随机变量的随机结构的一个规范。在统计学中，我们依赖对数据的分布情况做出假设，来对数据进行推断。我们需要一个公式来告诉我们分布中观测值出现的频率以及分布中的点如何取值。一个广为人知的分布是**正态分布**（也被称为**高斯分布**或**钟形曲线**）。我们喜欢将数据集与一个分布适配，因为如果数据集合理接近分布，就可以基于这个理论上的分布来对如何操作数据做出设想。

分布分为**连续型**和**离散型**。在离散分布中，数据只能取某些值。在连续分布中，数据可以是范围内的任何值。连续分布的一个例子是正态分布，离散分布的一个例子是二项分布。

正态分布允许我们假设统计抽样分布（例如样本平均值）在指定条件下呈正态分布。正态分布（参见图 1-5），或者叫**高斯分布**，是以 18 世纪数学家和物理学家卡尔·高斯的名字命名的。正态分布由其平均值和标准差来定义，在所有变化中通常具有相同的形状。

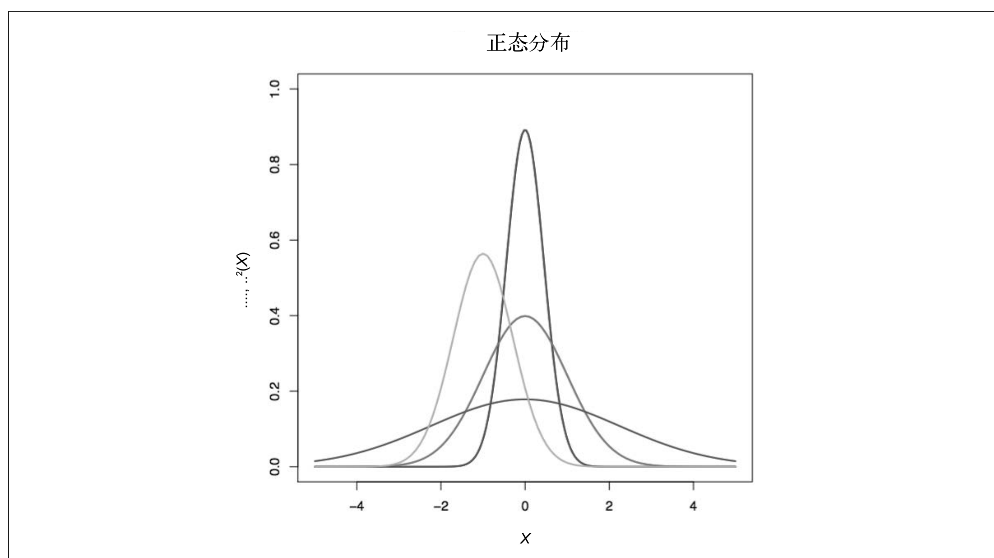


图 1-5：正态分布的例子

机器学习中其他的相关分布包括：

- 二项分布
- 逆高斯分布
- 对数正态分布

机器学习中训练数据的分布对于理解如何向量化建模数据很重要。

中心极限定理

如果样本足够大，样本平均值的抽样分布近似正态分布。不管样本所属总体怎样分布，这一点都成立。

基于这一事实，可以使用基于平均值的近似正态性的检验进行统计推断。即使样本是从非正态分布的总体中抽取的，这一点依然成立。

在计算机科学中，这个特性被应用于使用算法从非正态分布的总体中重复抽取指定大小样本的场景。当绘制从正态分布的总体中抽取的样本的直方图时，可以看到这种特性的作用。

长尾分布（如Zipf分布、幂律分布以及帕累托分布）表示一种高频总体后面跟着低频总体，且整体呈逐渐减少趋势的场景。这些分布由Benoit Mandelbrot在20世纪50年代发现，后来因作家克里斯·安德森的著作《长尾理论：为什么未来的商业销售更多小众商品》而广为人知。

例如，对零售商销售的商品进行排名，其中一些商品非常受欢迎，但大量商品的销量相对较少。这种排序频度分布（主要是受欢迎程度或“销量”）经常形成幂律。从这个角度来看，可以认为它们是长尾分布。

我们看看以下场景中的长尾分布。

- **地震损失**
  随着地震规模的增大，损失也越来越严重，所以损失程度不同。
- **粮食产量**
  有时会出现超出历史记录的产量，而模型倾向于围绕平均值调整。
- **预测从ICU病房出来之后死亡的概率**
  会有很多远远超出ICU病房中发生范围的、影响死亡率的事件。

这些例子在本书的分类问题背景下是相关的，因为大多数统计模型依赖于从大量数据中进行推断。如果更有趣的事件发生在分布的尾部，并且训练样本数据中没有包含这种情况，那么模型的表现也许不可预测。这种效应会在神经网络这样的非线性模型中增强。这种情况是“样本内/样本外”问题的特殊情况。即使经验丰富的机器学习实践者也会惊讶于模型在不全面的训练数据样本上表现得非常好，却不能泛化到更大的数据总体上。

遵循长尾分布的事件，其实际发生的可能性是标准偏差的五倍。必须注意在训练数据中适当选取事件，防止过拟合训练数据。稍后谈到过拟合以及第4章介绍调优时，将给出这种做法的更多细节。

1.4.5 样本与总体

数据总体被定义为希望在实验中研究或建模的所有单元。比如将研究的总体定义为“田纳西州所有的Java程序员”。

数据样本是数据总体的子集，我们希望它能代表数据的精确分布，而不会引入抽样偏差（例如对总体抽样的具体做法导致样本分布偏离）。

1.4.6 重采样方法

Bootstrapping和**交叉验证**是统计学中两种常用的重采样方法，对机器学习实践者很有用。

机器学习中的 Bootstrapping 是指从另一个样本中抽取随机样本来生成一个新样本，该样本在每个类别的样本数之间保持平衡。在对类别高度不平衡的数据集建模时，这很有用。

交叉验证（也被称为**轮换估计**）是评估模型对训练数据集泛化效果的一种方法。在交叉验证中，我们将训练数据集分成 n 个分片，然后将这些分片划分为训练组和测试组。使用训练组分片进行训练，然后用测试组分片进行测试。多次轮换两组之间的分片，直到用完所有组合。对要用到的分片数量没有强制规定，但研究人员在实践中发现 10 个分片的效果很好。经常也会单独留存一部分数据，用作训练时的验证数据集。

1.4.7 选择性偏差

选择性偏差指处理的采样方法没有适当的随机化，导致样本偏斜，不能代表想要建模的总体。在对数据集重采样时，需要意识到选择性偏差的存在，这样就不会在模型中引入偏差，否则将降低模型在更大的总体上的准确度。

1.4.8 似然

当讨论事件发生的可能性，但没有具体提到其概率数值时，使用非正式术语**似然**。一般来说，使用这个术语时，谈论的是一个事件，它有一个合理的发生概率，但也可能没有。也许有一些尚未被观察到的因素也会影响事件。在非正式场合，似然也被用作“概率”的同义词。

1.5 机器学习如何工作

关于求解线性方程组的前一节介绍了求解 $\boldsymbol{Ax}=\boldsymbol{b}$ 的基础知识。本质上，机器学习基于算法技术，通过**优化**来最小化这个方程的误差。

优化专注于改变 $\boldsymbol{x}$ 列向量（参数向量）中的数字，直到找到一组好的值，来得到与实际值最接近的结果。权重矩阵中的每个权重都会在损失函数计算了由网络产生的误差（基于实际结果，如先前所示的 $\boldsymbol{b}$ 列向量）之后被调整。将损失的某一部分归因于每个权重的误差矩阵将被乘以权重本身。

本章稍后将讨论 SDG，它是实现机器学习优化的主要方法之一。随着本书内容的推进，这些概念会与其他优化算法联结起来。也会介绍诸如正则化和学习率等关于超参数的基础知识。

1.5.1 回归

回归指的是试图预测实际输出值的函数，它根据自变量来估计因变量。最常见的回归类型是**线性回归**，它基于先前在线性方程组建模中所描述的概念。线性回归试图给出描述 x 和 y 之间关系的函数，然后对于已知的 x 值，准确预测 y 值。

1. 建立模型

线性回归模型的预测是系数（来自参数向量 $\boldsymbol{x}$）和输入变量（来自输入向量的特征）的线性组合。可以用下面的方程来模拟这个问题：

$$y = a + B\boldsymbol{x}$$

其中 a 是 y 轴截距，B 是输入特征，$\boldsymbol{x}$ 是参数向量。

这个方程可以扩展为以下形式：

$$y = a + b_0*\boldsymbol{x}_0 + b_1*\boldsymbol{x}_1 + \cdots + b_n*\boldsymbol{x}_n$$

线性回归求解问题的一个简单例子是根据通勤距离预测每月的汽油消耗量。在这个场景中，汽油成本是通勤距离的函数。汽油成本是因变量，而通勤距离是自变量。记录这两个量的关系，然后定义一个函数，比如：

$$成本 = f(距离)$$

这使得我们能够合理地基于里程预测汽油消耗量。在这个例子中，将距离作为自变量，成本是模型 f 中的因变量。

以下是线性回归模型的其他例子：

- 将体重作为身高的函数，以此来预测体重；
- 根据房屋的面积预测其销售价格。

2. 线性回归可视化

可以将线性回归表示为寻找一条尽可能接近数据散点图中很多点的直线，如图 1-6 所示。

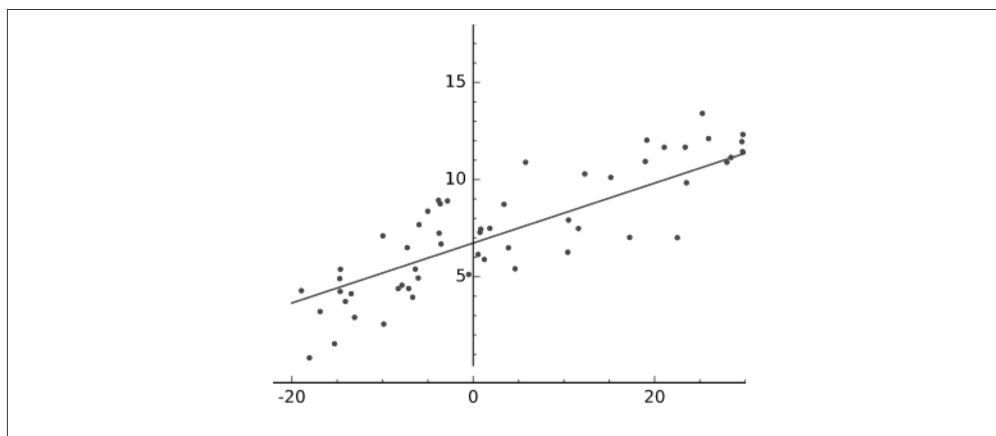


图 1-6：线性回归示意图

拟合就是定义函数 $f(x)$，它产生接近测量的 y 值或真实世界 y 值的 y 值。由 $y=f(x)$ 产生的直线接近因变量和自变量数值对的分散坐标。

3. 与线性回归模型联系起来

可以将该函数与之前的方程 $\boldsymbol{Ax}=\boldsymbol{b}$ 联系起来，其中 $\boldsymbol{A}$ 是要建模的所有输入示例的特征（例如，“权重”或“面积”）。每个输入记录是矩阵 $\boldsymbol{A}$ 中的一行。列向量 $\boldsymbol{b}$ 是矩阵 $\boldsymbol{A}$ 中所有输入记录的输出。使用误差函数和优化方法（例如 SGD），可以找到一组 $\boldsymbol{x}$ 参数，使得所有预测相对于真实结果的误差最小。

如同前面所讨论的，应用 SGD 时有三个组件来求参数向量 $\boldsymbol{x}$。

- **关于数据的假设**
  参数向量 $\boldsymbol{x}$ 和输入特征的内积（如上文所示）。
- **成本函数**
  预测的平方误差（预测 – 实际）。
- **更新函数**
  平方误差损失函数的导数（成本函数）。

线性回归处理直线，非线性的曲线拟合处理所有其他的曲线，尤其是那些 $\boldsymbol{x}$ 的指数大于 1 的曲线（这就是为什么有时机器学习被描述为“曲线拟合”）。一个绝对的拟合将穿过散点图上的每个点。不过讽刺的是，绝对拟合通常是非常差的结果，因为这意味着模型在训练集上训练得太完美了，如先前讨论过的，它对其未训练过的数据几乎没有预测能力（比如不能很好地泛化）。

1.5.2 分类

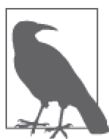
分类模型基于某个输入特征集为输出划分类别。如果说回归给出的结果是“多少”，那么分类给出的结果是“哪一种”。因变量 y 是类别型而非数值型。

最基本的分类形式是二元分类，它只有一个带有两个标签的输出（两个类别，分别为 0 和 1）。输出也可以是 0.0 到 1.0 之间的浮点数，以便处理没有绝对确定性的分类。在这种情况下，需要确定一个划分两个类别的阈值（通常为 0.5）。在文献中这些分类通常被称为阳性分类（如 1.0）和阴性分类（如 0.0），1.7 节将详细讨论。

二元分类的例子包括：

- 区分某人是否患有某种疾病；
- 区分电子邮件是否为垃圾邮件；
- 区分交易是否为欺诈或虚假交易。

除了二元分类，还有具有 n 个类别的分类模型，我们可以对每个输出类别评分，得分最高的类别是输出类别。当探讨具有多个输出的神经网络与具有单个输出（二元分类）的神经网络时将进一步讨论这一点。本章稍后探讨逻辑回归以及深入探讨神经网络的完整架构时，还会进一步讨论分类。



推荐

推荐是基于与用户相似的其他用户或用户以前浏览过的其他物品，向系统用户推荐物品的过程。因 Amazon.com 而声名大噪的协同过滤就是一种著名的推荐算法。

1.5.3 聚类

聚类是一种无监督学习技术，它通过计算距离，迭代地将相似的项更紧密地归类到一起。在过程结束时，最紧密地聚集在 n 个中心点附近的项被认为分类到该组。K-means 聚类是机器学习中一种著名的聚类算法。

1.5.4　欠拟合与过拟合

如同前面所提到的，优化算法首先试图解决欠拟合问题，即选取一条不能很好地逼近数据的直线，然后使之更好地逼近数据。横切弧形散点图的直线是欠拟合一个很好的例子，如图 1-7 所示。

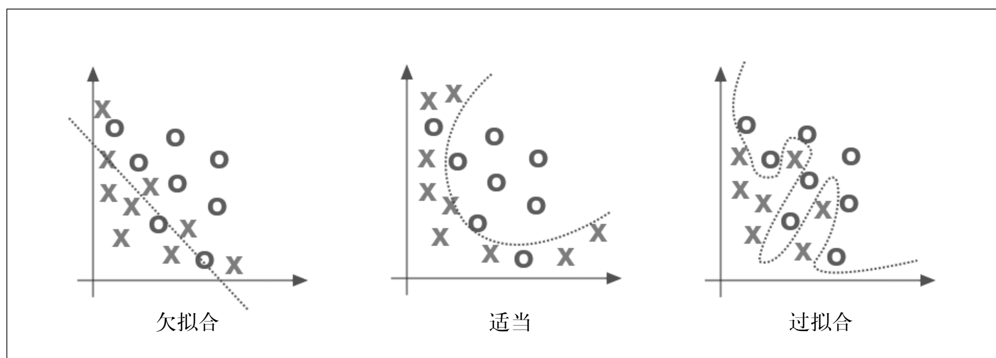


图 1-7：机器学习中的欠拟合与过拟合

如果这条线过于拟合数据，就会带来相反的烦恼，叫作“过拟合”。解决欠拟合是要优先考虑的问题，但是在机器学习中却要花费大量的精力来避免让线过拟合数据。当说模型过拟合数据集时，指的是它在训练数据上的误差率可能较低，但并不能很好地泛化到我们感兴趣的数据总体。

另一种解释过拟合的方法是考虑数据的可能分布。我们试图在其中画线的训练数据集只是更大的未知数据集的一个样本，如果需要所画的线具有预测能力，就需要它同样能较好地拟合更大的数据集。因此，必须假设样本松散地代表更大的集合。

1.5.5　优化

上述调整权重以对数据做出越来越准确的猜测的过程称为**参数优化**。可以将这个过程看作一种科研方法。先提出一个假设，在现实中检验它，然后不断改进或替换这个假设，以更好地描述世界上的事件。

每一组权重都代表一个关于输入意味着什么的假设，即它们如何与一个标签的含义相关。权重代表对网络输入和想要猜测的目标标签之间相关性的猜想。所有可能的权重和它们的组合可以被描述为这个问题的假设空间。试图提出最好的假设就是一个在假设空间中搜索，而我们使用误差和优化算法来实现。输入参数越多，问题的搜索空间就越大。学习的大量时间要花在决定哪些参数要忽略，哪些要保留上。



决策边界和超平面

当提到“决策边界”时，讨论的是由线性模型的参数向量所生成的 n 维超平面。

通过测量成本（即与真实数据点的距离）来用线拟合数据是机器学习的中心思想。这条线应该大体上拟合数据，这通过最小化线与所有点的距离之和来实现。将线上的点 x 与其对应的目标点 y 之间的距离之和调整到最小。在三维空间中，你可以想象山坡和山谷的差距，并把算法想象成一位摸索着斜坡行走的盲人旅行者。优化算法，如梯度下降，会告诉旅行者哪个方向是下坡，这样他就知道该往哪里走。

我们的目标是找到这样的权重，它能使网络的预测值（$\hat{b}$，或者 $\boldsymbol{A}$ 和 $\boldsymbol{x}$ 的点积）与测试集所知的真实值（$\boldsymbol{b}$）之间的差异最小，正如之前在图 1-4 中所看到的。上面的参数向量（$\boldsymbol{x}$）就是要找的权重。网络的准确度是它的输入和参数的函数，而使它变准确的速度是它的超参数的函数。



超参数

在机器学习中，既有模型参数，又有能够使网络更好、更快地训练的调优参数。这些调优参数称为**超参数**，在使用学习算法进行训练时，它们负责控制优化函数和模型的选择。



收敛

收敛指找到参数向量值的优化算法，它给出优化算法在所有训练样本中可能出现的最小误差。在优化算法尝试几个不同的参数之后，就称它在该解上迭代“收敛”。

以下是机器学习优化中的三个重要概念。

- **参数**
  转换输入以帮助确定网络推断的分类。
- **损失函数**
  在每次迭代过程中，评估分类（最小化误差）的效果。
- **优化函数**
  引导它走向最小误差点。

接下来仔细研究一个优化的子类——凸优化。

1.5.6 凸优化

在**凸优化**中，学习算法处理凸成本函数。如果 x 轴代表权重，y 轴表示成本，那么在 x 轴上某一点处的成本将下降到 0，而当权重在两个维度上偏离其理想值时，两侧的成本呈指数式上升。

图 1-8 显示了也可以反过来考虑成本函数。

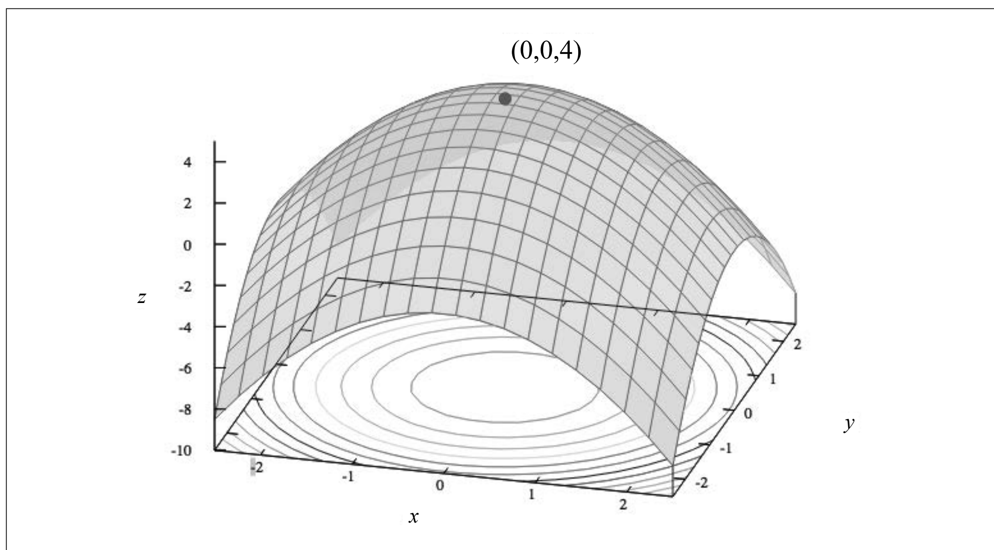


图 1-8：凸函数的可视化

另一种将参数与数据关联的方法是最大似然估计（maximum likelihood estimation，MLE）。MLE 研究一条抛物线，其开口向下，纵轴为似然值，横轴为参数。抛物线上的每一点代表给定一组参数时数据的似然值。MLE 的目标是对可能的参数进行迭代，直到找到使给定数据最有可能的集合。

从某种意义上说，最大似然和最小成本是同一枚硬币的两面。计算两个权重相对于误差的成本函数（这使我们处于三维空间），就像是一张固定了每个角的床单，中间下垂向外凸出——一种碗形的函数。这些凸曲线的斜率提示算法下一步该朝哪个方向走，正如我们在接下来探讨的梯度下降优化算法中所看到的一样。

1.5.7 梯度下降

在梯度下降的场景中，可以把网络预测的质量（权重 / 参数值的函数）想象为一幅风景画。小山代表预测误差很大的位置（参数值或权重），山谷表示误差较小的位置。我们选择风景画上的一个点作为初始权重。可以基于领域知识选择初始权重（如果正在训练一个对花卉进行分类的网络，那么花瓣的长度重要，但颜色不重要）。假如让网络做所有工作，我们就可以随机选择初始权重。

我们的目的是尽可能快地把权重降到误差较小的区域。像梯度下降这样的优化算法可以计算出山坡对于每个权重的实际坡度，即它知道哪个方向向下。梯度下降法测量坡度（由权重变化引起的误差变化），并将权重朝山谷底部移动一步。它通过求损失函数的导数来求梯度。在优化算法中，梯度给出算法下一步移动的方向，如图 1-9 所示。

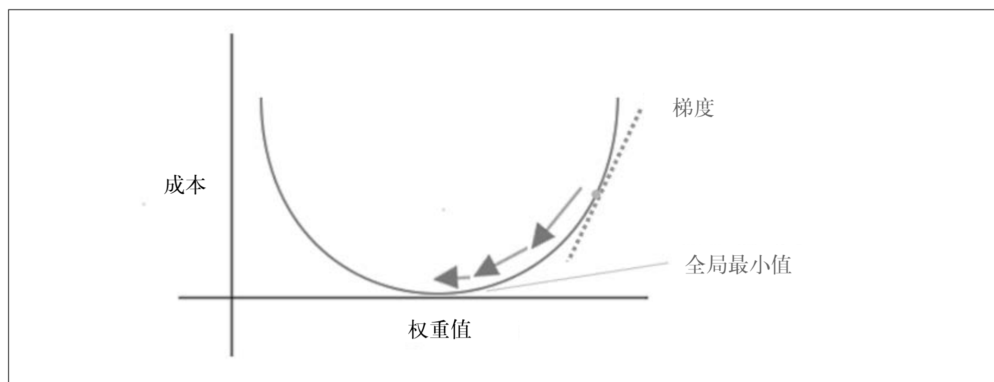


图 1-9：SGD 中朝全局最小值移动时的权重变化

导数用来衡量一个函数的“变化率”。在凸优化中，寻找函数导数等于 0 的点。这个点也被称为函数的**驻点**或**最小点**。在优化时，考虑以最小化函数（反转成本函数的外部）的方式优化函数。

什么是梯度

梯度被定义为一维函数的导数在多维函数 f 上的泛化，表示为函数 f 的 n 个偏导数的向量。这对优化很有用，因为在该函数最大增长率方向上的梯度值相当于图中该方向上的斜率。

梯度下降使用导数计算损失函数的斜率，读者应该很熟悉“导数”这个微积分概念。在二维损失函数上，导数是抛物线上任意点的正切，也就是 y 的变化除以 x 的变化，上升除以前进。

正如从三角学中所知，正切是一个比率，它等于直角三角形的对边（它测量垂直变化）除以邻边（它测量水平变化）。

曲线的一个定义是一条斜率不断变化的线，线上每个点的斜率等于紧贴该点的正切。由于斜率是由两个点求出的，那么如何精确地找到曲线上一个点的斜率呢？通过计算曲线上距离很小的两点之间连线的斜率，然后慢慢缩小距离直到接近零来求导数。在微积分运算中，这叫作**极限**。

不断重复计算误差并在低误差的方向修改权重的过程，直至权重到达误差最低的点，即准确度最高的点。使用凸损失函数（通常在线性建模）时，损失函数只有一个全局最小值。

可以从求参数向量 $\boldsymbol{x}$ 的三个组件的角度来思考线性建模。

- 一个关于数据的假设，例如“在模型中用于预测的方程”。
- 成本函数，也称损失函数，例如“误差平方和”。
- 更新函数；我们取损失函数的导数。

我们的假设是学习参数 x 和输入值（特征）的组合会给出一个分类或实际的输出值（回归）。成本函数告诉我们离损失函数的全局最小值有多远，我们用损失函数的导数作为更新函数来改变参数向量 x。

损失函数的导数给出 x 中的每个参数需要调整的幅度，以便更接近损失曲线上的 0 点。本章稍后将更详细地研究这些方程，展示它们如何用于线性回归和逻辑回归（分类）。

然而在其他非线性问题中，并不总是能够得到这样一条干净的损失曲线。这些非线性的、假想的风景画的问题是：可能有多个山谷，但是通过梯度下降来求更低权重的机制却无法知道它是否已经到达最低谷，或者仅仅是在一个较高的山谷的最低点。最低谷的最低点被称为**全局最小值**，而其他所有山谷的最低点称为**局部最小值**。梯度下降法有可能陷入局部最小值，这是它的一个缺点。第 6 章探讨超参数和学习率时将介绍解决这一问题的方法。

梯度下降面对的第二个问题是非规范化特征。本书中的“非规范特征”指需要用差别很大的比例尺来测量的特征。如果有一个维度是百万级的，而另一个维度是小数级的，那么梯度下降将很难找到最陡峭的斜率来使误差最小化。



规范化处理

第 8 章将深入研究向量化语境下的规范化方法，并介绍一些更好地解决这一问题的方法。

1.5.8 SGD

在梯度下降中，在计算梯度和更新参数向量之前，计算所有训练样本的总损失。而在 SGD 中，在每次训练样本之后计算梯度和更新参数向量。这已被证明可以加快学习速度，同时也有利于并行，本书后面详细讨论。SGD 是“全批量”梯度下降的近似值。

小批量训练与SGD

SGD 的另一个变体是使用多个训练样本计算梯度，但不会使用整个训练数据集。这种变体被称为**小批量** SGD 训练，它已被证明比仅使用单个训练实例的做法性能更好。应用小批量 SGD 也会使收敛更平滑，因为每次迭代都会使用更多的训练样本来计算梯度。

随着小批量大小的增加，计算出的梯度会更接近整个训练集的“真实”梯度，这也带来了更高的计算效率。如果设置的小批量过小（例如，一条训练记录），就无法有效地使用硬件，尤其是在像 GPU 可用这样的情况下。相反，如果小批量过大（超过一定界限）就会导致低效，因为可以用普通的梯度下降方法和更少的计算量（在某些情况下）求得同样的梯度。

1.5.9 拟牛顿优化方法

拟牛顿优化方法是迭代算法，涉及一系列的“线性搜索”。它与其他优化方法的区别在于搜索方向的选择方式。本书后面几章将进一步探讨这些方法。

雅可比矩阵和海森矩阵

雅可比矩阵是一个包含函数对各向量的一阶偏导数的 $m \times n$ 矩阵。

海森矩阵是一个函数的二阶偏导数的方阵，描述拥有许多变量的函数的局部曲率。海森矩阵在使用牛顿型方法解决大规模优化问题中得到了应用，因为它们是局部泰勒展开式的二次项系数。在实践中，海森矩阵很难计算，我们往往使用拟牛顿算法近似地替代海森矩阵。这类拟牛顿优化算法的一个例子是 L-BFGS，第 2 章将详细介绍。

本书不会过多提及雅可比矩阵和海森矩阵，但是希望你能了解它们及其在更广阔的机器学习领域中的地位。

1.5.10 生成模型与判别模型

使用不同类型的模型会生成不同类型的输出。两个主要的模型类型是**生成模型**和**判别模型**。生成模型根据数据的创建方式生成相应类型的响应或输出。判别模型不关心数据的创建方式，只是简单地根据给定的输入信号给出分类或类别。判别模型侧重于对类别之间的边界建模；与生成模型相比，它能够更细微地描述边界。判别模型通常用于机器学习中的分类。

生成模型学习**联合**概率分布 $p(x, y)$，而判别模型学习**条件**概率分布 $p(y|x)$。分布 $p(y|x)$ 是接收输入 x 并产生输出（或分类）y 的自然分布，因此得名“判别模型”。生成模型学习分布 $p(x, y)$，被用来根据给定的输入产生可能的输出。生成模型通常被设置为捕捉数据中微妙关系的概率图模型。

1.6 逻辑回归

逻辑回归是线性建模中一种著名的分类方法，适用于二元分类以及多项式逻辑回归形式的多个标签分类。逻辑回归是一种回归模型（技术上讲），其中因变量是类别变量（例如“分类”）。二元逻辑回归模型用于基于一组（一个或多个）输入变量（自变量或“特征”）估计一个二元输出的概率。该输出是一个类别的统计概率，是基于给定的输入做出的预测。

与线性回归相似，可以用 $\boldsymbol{Ax}=\boldsymbol{b}$ 的形式表达逻辑回归建模问题，其中 $\boldsymbol{A}$ 是要建模的所有输入样本的特征（例如，“权重”或“面积”）。每个输入记录是矩阵 $\boldsymbol{A}$ 中的一行，列向量 $\boldsymbol{b}$ 是矩阵 $\boldsymbol{A}$ 中所有输入记录的结果。使用成本函数和优化方法，可以找到一组 $\boldsymbol{x}$ 参数，使所有预测值与真实结果的误差最小化。

再次使用 SGD 优化这个问题，我们有三个组件来解决参数向量 $\boldsymbol{x}$。

- **关于数据的假设**

$$f(x) = \frac{1}{1 + \mathrm{e}^{-\theta x}}$$

- **成本函数**
  “最大似然估计”
- **更新函数**
  成本函数的导数

在这种情况下，输入是自变量（例如，输入列或“特征”），而输出是因变量（例如，“标签分数”）。可以简单地理解为：逻辑回归函数根据输入值与权重来判断一个结果是否可能。接下来仔细研究逻辑函数。

1.6.1 逻辑函数

在逻辑回归中，逻辑函数（“假设”）定义如下：

$$f(x) = \frac{1}{1 + e^{-\theta x}}$$

这个函数在逻辑回归中很有用，因为它可以接纳从负无穷到正无穷范围内的任何输入，并且将输入映射到 0.0 到 1.0 范围内，这使输出值可以解释为概率。图 1-10 就是一个逻辑函数方程的图像。

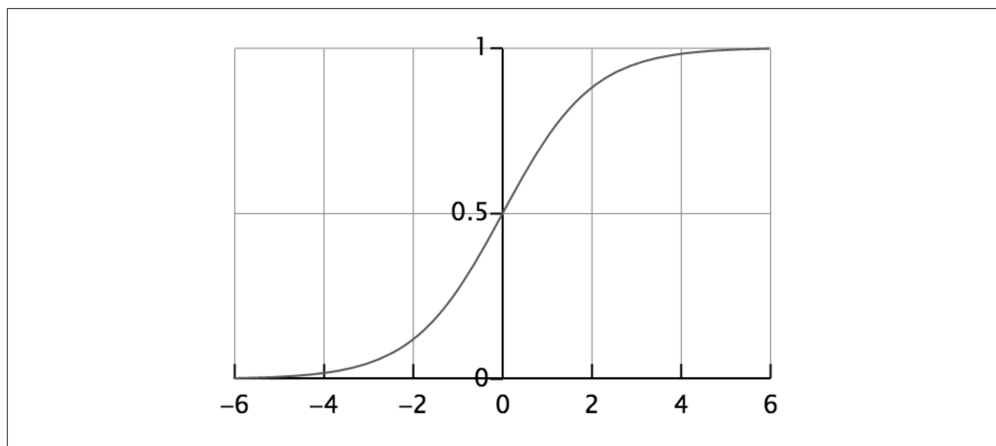


图 1-10：逻辑函数示意图

这个函数是**连续对数 sigmoid 函数**，它的范围是 0.0 到 1.0，我们将在 2.3 节中再次见到这个函数。

1.6.2 理解逻辑回归的输出

逻辑函数通常用希腊字母 Σ 或 σ 表示，因为 x 和 y 的关系在二维图上看起来像一个拉长的、风吹过的字母 s，它的最大值和最小值分别逼近 1 和 0。

如果 y 是 x 的函数，并且该函数是 sigmoid 函数或逻辑函数，那么 x 越大，我们就越接近

1/1，因为 e 的负无穷次幂趋近于 0；相反，x 相比 0 越小，表达式 $(1+e^{-\theta x})$ 的值越大，整个商就越小。因为 $(1+e^{-\theta x})$ 是分母，所以它越大，商本身就越接近零。

逻辑回归中的 $f(x)$ 表示对于每个给定的输入 x，y 等于 1（即为真）的概率。如果要预估一封电子邮件是垃圾邮件的概率，而 $f(x)$ 恰好等于 0.6，则可以说对于给定的输入，y 有 60% 的概率为 1，或者说电子邮件有 60% 的可能性是垃圾邮件。如果将机器学习定义为从已知输入推断未知输出的一种方法，那么逻辑回归模型中的参数向量 $\boldsymbol{x}$ 决定了推断的确定性。



logit 变换

logit 函数是逆逻辑函数（“逻辑变换”）。

1.7 评估模型

评估模型是了解模型在特定情况下给出正确分类结果的准确程度，以及评估预测值的过程。有时候，我们只关心模型做出正确预测的频率。其他时候，模型能比其他模型更准确地做出某一类预测很重要。本节将讨论不良阳性、无害阴性、类别不平衡和非均等代价等话题。接下来介绍用于评估模型的基本工具：**混淆矩阵**。

混淆矩阵

混淆矩阵（参见图 1-11）也称**混淆表**，是表示分类器的预测值和实际结果（标签）之间关系的、由行和列构成的表。在于适当的时候给出正确答案的基础上，我们使用这个表来更好地理解模型或分类器的表现。

	P′（预测值）	N′（预测值）
P（实际值）	TP（真阳性）	FN（假阴性）
N（实际值）	FP（假阳性）	TN（真阴性）

图 1-11：混淆矩阵

根据以下规则判断预测属于哪种类型。

- 真阳性
  - 预测为阳性
  - 标签为阳性
- 假阳性
  - 预测为阳性
  - 标签为阴性
- 真阴性
  - 预测为阴性
  - 标签为阴性
- 假阴性
  - 预测为阴性
  - 标签为阳性

在传统统计学中，假阳性也称“I 型错误”，而假阴性也称“II 型错误”。通过跟踪这些度量值，除了得到简单的正确率数据，还可以对模型性能做更详细的分析。根据前面混淆矩阵中四个值的组合，我们能够对模型做出不同角度的评估，如下所示：

```
Accuracy:  0.94
Precision: 0.8662
Recall:    0.8955
F1 Score:  0.8806
```

在上例中可以看到四种常见的评估机器学习模型的方法。之后会简要介绍每一种评估方法，但现在先了解评估模型灵敏度与特异度的基础知识。

1. 灵敏度与特异度

灵敏度和**特异度**是对二元分类模型的两种评估方法。真阳性率等于将一条输入记录分类为正向类并且它是正确分类的频率。这也被称为灵敏度或召回率。一个例子是将实际上确实患病的病人诊断为患病。灵敏度量化了模型避免假阴性的能力。

$$灵敏度 = TP / (TP + FN)$$

如果模型将前面例子中的一个病人分类为没有患病，而她实际上也确实没有患病，那么这种情况被称为真阴性（也称特异度）。特异度量化了模型避免假阳性的能力。

$$特异度 = TN / (TN + FP)$$

很多时候需要在灵敏度和特异度之间进行权衡。一个例子是建立一个模型来更频繁地检测病人的严重疾病，因为误诊一个真正患病的病人的成本很高。我们认为这种模型具有低特异度。一种严重的疾病可能危及病人和他周围其他人的生命，所以模型应对这种疾病及其影响有高灵敏度。完美的模型应当是 100% 灵敏（即所有疾病都被检测出来）和 100% 特异的（即没有一个未生病的人被分类为患病）。

2. 准确度

准确度是一个量的测量值与其真实值的接近程度。

$$准确度 = (TP + TN) / (TP + FP + FN + TN)$$

当类别非常不平衡时，准确度可能会误导模型的质量。如果简单地将所有事物分类为较大的类别，模型将自动做出大量正确的猜测，并具有一个很高的准确度分数，但基于模型创建的实际应用却会误导判断（例如，它永远不会预测更小的类别或罕见的事件）。

3. 精度

在科学和统计学中，在相同的条件下重复测量得到相同结果的程度称为精度。精度也被称为**阳性预测值**。虽然有时在口语中它与“准确度”互换使用，但是它们在科学方法框架内的定义不同。

精度 = TP / (TP + FP)

测量可以准确但不精确，不准确但仍然精确，既不准确也不精确，或既准确又精确。如果一个测量既准确又精确，那么就认为它是有效的。

4. 召回率

这是与“灵敏度”相同的概念，也被称为**真阳性率**或**命中率**。

5. F1

在二元分类中，用 F1 分数（或 F 分数、F 测度）作为模型准确度的度量。F1 分数将精度和召回率（之前介绍的）合二为一，是二者的调和平均值，其定义如下所示。

F1 = 2TP / (2TP + FP + FN)

F1 的分数在 0.0 到 1.0 之间，其中 0.0 是最差的分数，1.0 是我们希望看到的最好分数。F1 分数通常用在信息检索中，用于了解模型检索相关结果的表现如何。在机器学习中，F1 分数被用作模型表现的总体得分。

6. 场景与分数的解释

如本节前面所介绍的，场景会影响模型的评估方式，也会决定何时使用不同类型的分数。类别不平衡对评估分数的选择有很大的影响，在许多数据集中，我们会发现类别或标签的数量不均衡。以下是一些典型的领域：

- 网页点击预测；
- ICU 死亡率预测；
- 欺诈检测。

在这些场景中，完全追求“百分比至上”的得分可能会损害模型的实用价值。这种问题的一个例子是 2012 年 PhysioNet 挑战赛的数据集。

这个挑战的目标是“使用二元分类器以最大准确率预测院内死亡率”。对这个数据集建模的困难和挑战在于：预测患者存活是很容易的，因为数据集中的大量样本都有患者存活的结果。在这种场景下，目标是准确预测死亡率，这是在现实世界的临床相关场景中模型最有价值的地方。在这个竞赛中，分数以如下方式计算：

分数 = MIN(精度, 召回率)

这样设定分数，是为了让参赛者不仅仅专注于预测患者大部分时间将存活并获得良好的 F1 评分，而是专注于预测患者何时会死亡（始终专注在临床相关上）。这是场景会影响模型评估方法的一个很好的例子。



处理类别不平衡的方法

第 6 章将介绍解决类别不平衡的实用方法。我们将在分类和回归的场景下，更细致地从不同角度研究类别不平衡和误差分布。

1.8 建立对机器学习的理解

本章介绍了为实践机器学习所需了解的核心概念。我们围绕下面的方程，回顾了建模时要用到的核心数学概念。

$$\boldsymbol{Ax} = \boldsymbol{b}$$

本章还研究了将特征转换为矩阵 $\boldsymbol{A}$ 的核心思想，改变参数向量 $\boldsymbol{x}$ 以及在向量 $\boldsymbol{b}$ 中设置结果的方法，还介绍了一些改变参数向量 $\boldsymbol{x}$，使目标函数的分数（或“损失”）最小的基本方法。

本书后面的章节将继续扩展这些关键概念，介绍神经网络和深度学习如何在这些基本原理的基础上，使用更复杂的方法来创建矩阵 $\boldsymbol{A}$，通过优化方法修改参数向量 $\boldsymbol{x}$，并在训练期间测量损失。接下来进入第 2 章，这一章将在神经网络的基础上进一步介绍这些概念。

第 2 章

神经网络基础与深度学习

2.1 神经网络

神经网络是一种计算模型，它与动物的大脑有一些共性：其中许多简单的单元能够并行工作，并不需要中央控制单元协调。单元之间的权重是神经网络中长期存储信息的主要手段，更新权重是神经网络学习新信息的主要方式。

第 1 章讨论了 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ 形式的方程组的建模。在神经网络领域，矩阵 $\boldsymbol{A}$ 仍是输入数据，列向量 $\boldsymbol{b}$ 仍是矩阵 $\boldsymbol{A}$ 中每一行的标签或结果。神经网络连接的权重是 $\boldsymbol{x}$（参数向量）。

神经网络的行为由其网络架构决定。网络架构（部分）定义如下：

- 神经元数量；
- 层数；
- 层之间的连接类型。

最著名且最容易理解的神经网络是前馈多层神经网络。它有一个输入层、一个或多个隐藏层和一个输出层。每层有不同数量的神经元，并且每层完全连接到相邻层。层中神经元之间的连接形成无环图，如图 2-1 所示。

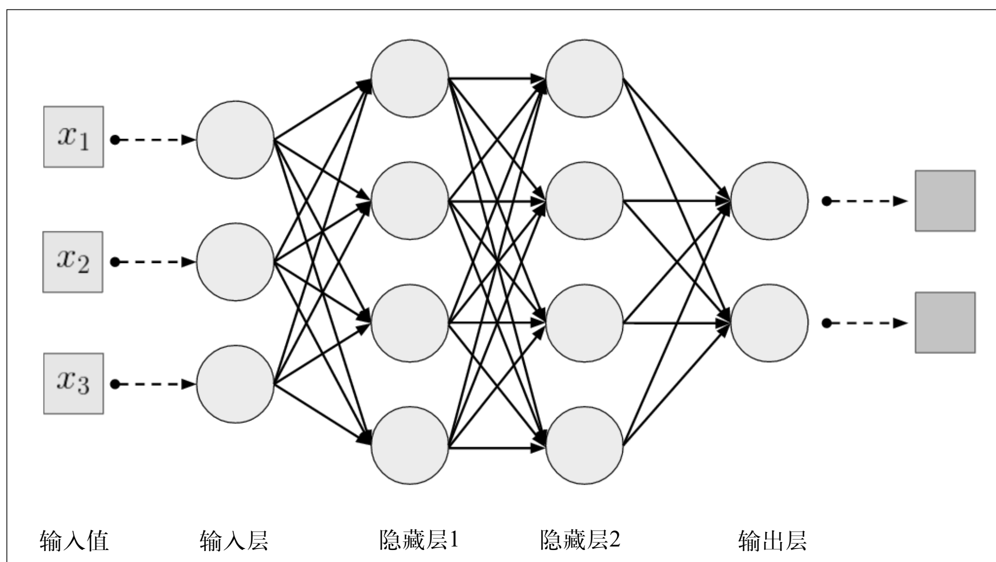


图 2-1：多层神经网络拓扑

只要给定足够的人工神经元单元，前馈多层神经网络就可以表示任何函数。它通常使用被称为**反向传播学习**的学习算法进行训练。反向传播在神经网络中的连接权重上使用梯度下降（参见第 1 章），以此最小化网络输出的误差。



局部最小值与反向传播

反向传播可能会陷入局部最小值，但在实践中通常表现良好。

历史上，人们一直认为反向传播很缓慢，但最近由于并行计算和图形处理单元（GPU）带来了计算能力的提升，人们对它重新燃起了兴趣。

在互联网和文献中有许多对神经网络与人类思维之间联系的自以为是的见解，我们需要从巨大的噪声中明辨真知。下面介绍人工神经网络从生物学获取的灵感。



思维的机械论观点

比起构建一棵需要明确所有输入的严格定义的树，我们可以构建一个反映外部世界发送给我们的部分且模糊的信息的模型，从中做出相对而不是完全确定的推断。

神经网络的这个特点代表了它与 20 世纪初占主导地位的思维的机械论观点的决裂，这种观点认为我们的大脑以一种确定性的方式与世界联系在一起（就像两个啮合在一起的齿轮），它接受明确的输入，从而产生明确的输出。现在我们认为，人类是基于不完整的，甚至有时是矛盾的信息，找到前进和行动的方法的。人脑基于概率进行推断，神经网络也是如此。

我们将简单回顾生物神经元，然后介绍现代神经网络的前身：**感知器**。基于对感知器的理解，我们将看到它是如何演变为支撑现代前馈多层感知器的、更通用的人工神经元的。本章旨在为现代神经网络实践者深入挖掘更多奇妙的深度网络架构打下基础。

2.1.1 生物神经元

生物神经元（见图 2-2）是为所有动物的神经系统提供基本功能单元的神经细胞。神经元相互沟通，通过突触将电化学脉冲从一个细胞传递到下一个细胞，只要脉冲强到足以突破突触间隙，能够激活化学物质的释放。脉冲的强度必须超过最小阈值，否则化学物质不会被释放。

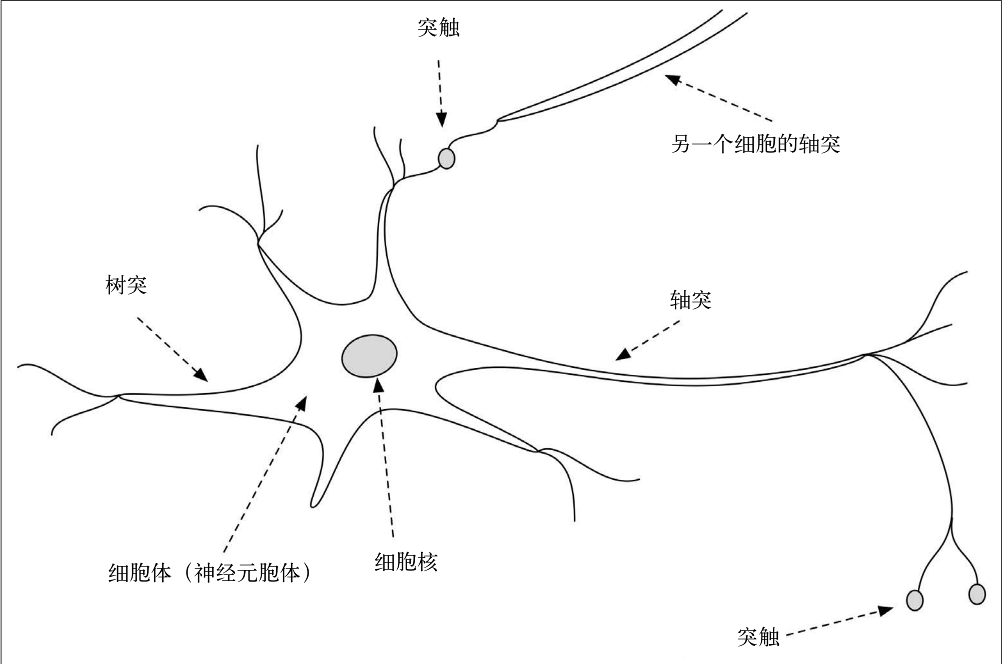


图 2-2：生物神经元

图 2-2 展示了神经细胞的主要部分：

- 神经元胞体
- 树突
- 轴突
- 突触

神经元由一个神经细胞构成，而神经细胞由一个神经元胞体（细胞体）构成，这种胞体有许多树突，但只有一个轴突，然而单个轴突可以分支数百次。树突是从主细胞体突出来的细长结构。轴突是从细胞体延伸出来的具有特殊扩展的神经纤维。

1. 突触

突触是轴突与树突之间的连接点。大多数突触从一个神经元的轴突向另一个神经元的树突

发送信号。例外的情况是神经元可能缺少树突，或缺少轴突，又或者缺少连接两个轴突的突触。

2. 树突

在神经细胞周围浓密的网络中，树突的纤维从神经元胞体延伸出来。树突允许细胞从邻接的神经元接收信号，每个树突能够乘以它的权重值。这里做乘法意味着突触神经递质与引入树突的化学信号的比率的增大或减小。

3. 轴突

轴突是从主神经元胞体延伸出来的单个长纤维。其延伸的距离比树突长，长度一般为 1 厘米（神经元胞体直径的 100 倍）。最终，轴突将分叉并连接到其他树突。神经元能够通过跨膜电压的变化发送电化学脉冲，从而产生**动作电位**。这个信号沿细胞的轴突传播，并激活与其他神经元的突触连接。

4. 跨生物神经元的信息流

增加电位的突触被认为是**刺激性的**，而那些降低电位的突触被认为是**抑制性的**。**可塑性**指因输入的刺激而产生的连接强度的长期变化。神经元也被证明会随着时间推移形成新的连接，甚至迁移。这些连接改变的机制推动了生物大脑中的学习过程。

5. 从生物神经元到人工神经元

动物的大脑已被证明是产生思想的基本组成部分。我们可以研究大脑的基本组成部分并了解它们。研究已经发现了标记大脑各部分功能并追踪信号在神经元中传递的方法。



CNN 与哺乳动物的视觉系统

本书后面的章节将介绍一个被称为 CNN 的深度网络。CNN 在不同层的图像表现与大脑处理视觉信息的做法相似。虽然这项研究很有趣，但并不意味着 CNN 是对哺乳动物大脑活动的完全模仿。

然而，我们仍然没有完全理解这种去中心化的功能单元集合是如何为思想和意识的产生提供基础的。



意识的产生地

18 世纪，人们开始认识到大脑才是“意识的产生地”。19 世纪晚期，动物大脑的各部分逐渐被标记出来，以便更好地理解其功能区域。之前人们认为意识的产生地是心脏，甚至还有人认为是脾脏。

了解了生物神经元工作原理的基础知识之后，接下来介绍人们对神经元建模的首次尝试：感知器的出现。

2.1.2 感知器

感知器是用于二元分类的线性模型。在神经网络领域，感知器被认为是使用单位阶跃函数作为激活函数的人工神经元，本章稍后将详细介绍这两个概念。感知器的前身是 McCulloch 和 Pitts 在 1943 年开发的阈值逻辑单元（TLU），它可以学习与（AND）和或

（OR）逻辑函数。感知器训练算法被认为是监督学习算法。随着下面的介绍，你会了解到 TLU 和感知器都受到了生物神经元的启发。

1. 感知器的历史

感知器是 Frank Rosenblatt 于 1957 年在康奈尔航空实验室发明的。它由美国海军研究办公室资助，并被《纽约时报》报道了：

> 电子计算机的雏形，（海军）期望它能行走、说话、看、写、复制自己，并意识到自己的存在。

显然这些预测有些草率，因为我们已经见过人们对机器学习和人工智能前景的夸大宣传。它的早期版本打算用物理机器而不是软件程序来实现。它的第一个软件实现是为 IBM 704 开发的，之后它在 Mark I 感知器中得到实现。

还应该了解的一点是，McCulloch 和 Pitts 在 1943 年还基于阈值和加权和引入了“神经活动分析”这个基本概念，它们是为之后的变体（如感知器）开发模型的关键。



Mark I 感知器

Mark I 感知器是美国海军为军事目的而设计的图像识别系统。Mark I 感知器有 400 个光电池连接到机器中的人工神经元，权重由电位器实现。权重更新由电动机实现。

2. 感知器的定义

感知器是一个线性模型的二元分类器，具有简单的输入 – 输出关系，如图 2-3 所示。我们将 *n* 个输入值与其相关权重相乘，然后求和，再将这个“净输入”发送到定义了阈值的单位阶跃函数。在感知器中，这通常是一个阈值为 0.5 的单位阶跃函数，它会根据输入，输出单个实数二元值（0 或 1）。

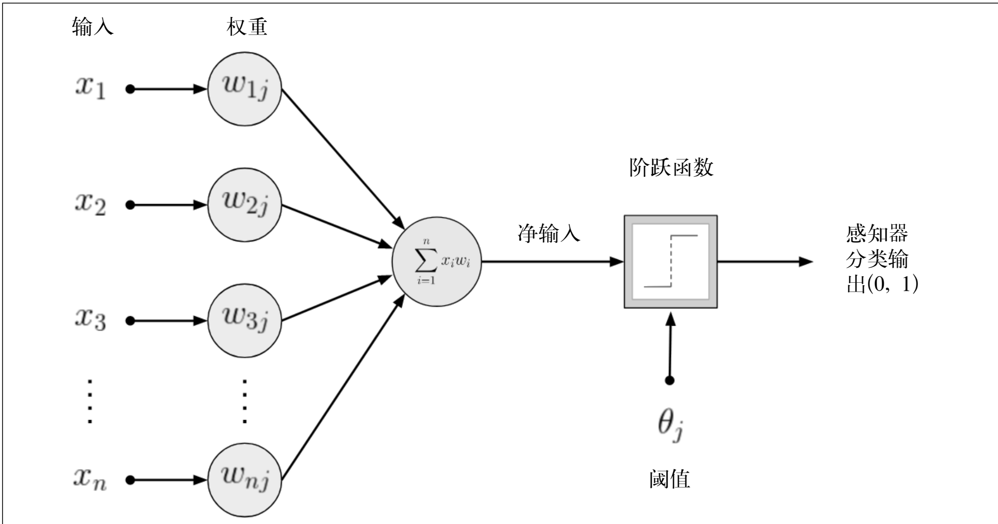


图 2-3：单层感知器

可以对单位阶跃函数方程中的决策边界和分类输出建模，如下所示。

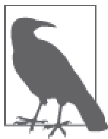
$$f(x)=\begin{cases}0 & x<0\\ 1 & x\geqslant 0\end{cases}$$

为了产生激活函数的净输入（这里是单位阶跃函数），取输入和连接权重的点积。在图 2-3 的左半部分可以看到，这些结果会作为求和函数的输入。表 2-1 展示了求和函数是如何计算的，还对求和函数的参数进行了说明。

表2-1：求和函数的参数

函数参数	说　明
$\boldsymbol{w}$	连接的实数权重向量
$\boldsymbol{w}\cdot\boldsymbol{x}$	点积（$\sum_{i=1}^{n} w_i x_i$）
n	感知器输入的个数
b	偏置项（输入值不影响它的值；它使决策边界偏离原点）

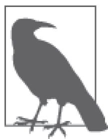
阶跃函数（激活函数）的输出是感知器的输出，并给出输入值的分类。如果偏置值为负，它将迫使学得的权重和为更大的值，以获得值为 1 的分类输出。偏置项移动模型的决策边界。输入值不影响偏置项，但偏置项通过感知器学习算法学得。



单层感知器

在神经网络的研究中，感知器更广泛的称谓是“单层感知器”，以区别于它的继任者“多层感知器”。

作为一个基本的线性分类器，单层感知器是前馈神经网络家族最简单的形式。



感知器与生物神经元的关系

虽然没有一个完整的模型来解释大脑是如何工作的，但我们确实看到感知器仿照了生物神经元。感知器采取与突触向其他生物神经元传递信息相似的方式，从与之相连的、有权重的连接获取输入。

3. 感知器学习算法

感知器学习算法改变感知器模型中的权重，直到所有输入记录都被正确分类。如果学习的输入不是线性可分的，算法就不会终止。线性可分的数据集指我们可以找到一个超平面的值，干净地将数据集划分为两类。

感知器学习算法在训练开始时用小的随机值或 0.0 初始化权重向量。正如我们在图 2-3 中看到的，感知器学习算法获取每个输入记录，之后计算输出分类，并检查它与实际的分类标签是否相同。为了产生分类，列（特征）将组成权重，其中 n 是输入与权重的维度。第一个输入值是偏置输入，它总为 1.0，因为我们不会影响偏置输入。在这个图中，第一个权重是偏置项。输入向量和权重向量的点积成为了激活函数的输入，正如之前所探讨的一样。

如果分类正确，我们不会调整权重。如果分类不正确，就需要相应地调整权重。通过“在线学习”的方式，更新各个训练样本之间的权重。这个循环一直持续到所有输入样本都被正确分类为止。如果数据集不是线性可分的，训练算法就不会终止。图 2-4 展示了一个线性不可分的异或逻辑函数的数据集。

x_0	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

图 2-4：异或函数

一个基本的感知器（单层感知器）不能解决异或逻辑建模问题，这体现了早期感知器模型的局限性。

4. 早期感知器的局限性

在尝到甜头之后，人们发现感知器能识别的模式类型有限。最初无法解决非线性（例如线性不可分的数据集）问题被视为神经网络领域的一个失败。Minsky 和 Papert 在 1969 年出版的《感知器》一书中阐释了单层感知器的局限性。然而业界还没有广泛认识到，在众多的非线性问题之中，多层感知器其实可以解决异或问题。



人工智能的寒冬 I：1974—1980

对多层感知器能力的误解是人工智能早期的一个巨大挫折，之后的十年人们降低了对神经网络的兴趣及资金投入。直到 20 世纪 80 年代中期，随着反向传播算法的流行（虽然反向传播算法最初在 1974 年由 Webos 提出），人工智能才再度复兴，神经网络开始了它的第二次浪潮。

2.1.3 多层前馈网络

多层前馈网络是具有一个输入层、一个或多个隐藏层和一个输出层的神经网络。每一层都有一个或多个人工神经元。这些人工神经元与它们的前身感知器相似，但根据网络中每一层特定目的的不同，具有不同的激活函数。本章稍后将更深入地探讨多层感知器中层的类型。接下来先仔细了解为解决单层感知器的局限性而出现的这个进化的人工神经元。

1. 人工神经元的进化

多层感知器的人工神经元类似于它的前身——感知器，但是它在可使用的激活层类型上增加了灵活性。图 2-5 是基于感知器的更新后的人工神经元示意图。

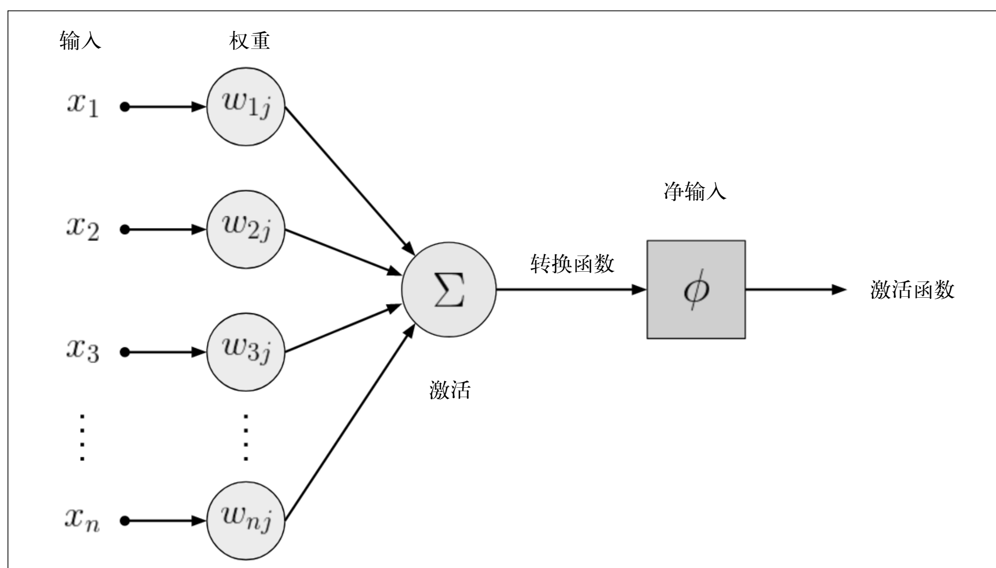


图 2-5：多层感知器的人工神经元

该示意图类似于图 2-3 的单层感知器，但图中使用的是更泛化的激活函数。之后将进一步拓展这张示意图，详细讲解人工神经元。



关于“神经元”一词的说明

从这里开始一直到本书的末尾，“神经元”这个词指基于图 2-5 的人工神经元。

激活函数的净输入仍然是权重和输入特征的点积，但灵活的激活函数允许我们根据它的输出值创建不同类型的输出。这是与之前使用分段线性单位阶跃函数的感知器设计的主要不同，这种改进使得人工神经元现在可以表示更复杂的激活输出。

人工神经元输入。人工神经元（见图 2-6）将输入值乘以连接上的权重，输入可以被忽略（通过输入连接上的 0.0 权重值控制）或者传给激活函数。如果不提供非零的激活值作为输出，那么激活函数还能过滤数据。

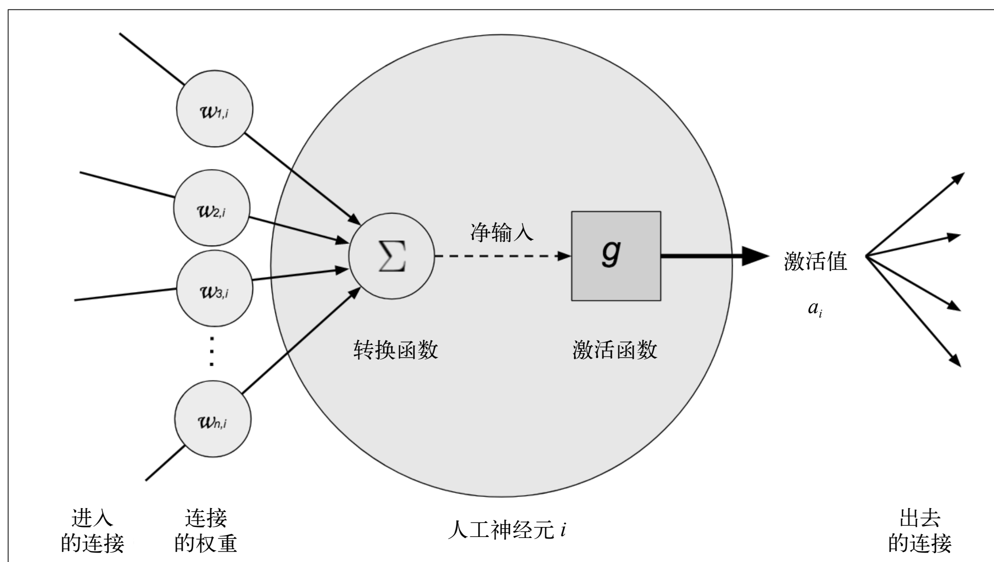


图 2-6：多层感知器神经网络中人工神经元的细节

如图 2-6 所示，神经元的净输入表示为连接上的权重乘以进入连接的激活值。对于输入层，我们只考虑特定索引的特征，并且激活函数是线性的（它传递特征值）。对于隐藏层，输入是来自其他神经元的激活值。在数学上，可以将人工神经元的净输入（总加权输入）表示为：

$$\text{input_sum}_i = \boldsymbol{W}_i \cdot \boldsymbol{A}_i$$

其中，$\boldsymbol{W}_i$ 是通向神经元 i 的所有权重的向量，而 $\boldsymbol{A}_i$ 是神经元 i 的输入的激活值的向量。我们在每一层增加偏置项，建立下面这个方程（稍后详细解释）：

$$\text{input_sum}_j = \boldsymbol{W}_i \cdot \boldsymbol{A}_i + b$$

为了从神经元产生输出，接着用一个激活函数 g 封装这个净输入，方程如下所示：

$$a_i = g\,(\text{input_sum}_i)$$

然后我们可以用 input_i 的定义展开这个函数：

$$a_i = g(\boldsymbol{W}_i \cdot \boldsymbol{A}_i + b)$$

输出的神经元 i 的激活值作为输入值，通过到其他人工神经元的连接（乘以连接上的权重）传递到下一层。



不同的表示方法

在研究论文中经常能见到下面这种人工神经元输出的替代表示方法：

$$h_{w,b}(x) = g(\boldsymbol{w} \cdot \boldsymbol{x} + b)$$

这种表示方法与前面的方程略有不同。这里展示这个替代的表示方法，是想说明一些论文会使用稍微不同的表示方法来解释这些概念，希望你能够识别这些变体。

如果激活函数是 sigmoid 函数，将得到：

$$g(z)=\frac{1}{1+\mathrm{e}^{-z}}$$

这个输出的范围是 [0, 1]，与逻辑回归函数的输出相同。



sigmoid 激活函数在实践中的应用

这里列出 sigmoid 激活函数是因为它曾经流行过，之后会介绍，sigmoid 激活函数的受欢迎程度已经大不如前。

输入是想从中产生信息的数据，连接权重和偏置是用于控制活动（激活或不激活）的数值。与感知器一样，有一种学习算法可用来改变每个人工神经元的权重和偏置值。在训练阶段，随着网络学习的进行，权重和偏差会发生变化。本章稍后会介绍神经网络的学习算法。

就像生物神经元不会将它们接收的每一个电化学脉冲都传递下去一样，人工神经元也不仅仅是传递信号的电线或二极管。它们是有选择性的。它们过滤接收到的数据，并只聚合、转换和传输特定的信息到网络中的下一个神经元。当这些过滤器和转换器作用于数据时，它们能在更大的多层感知器神经网络中将原始输入数据转换为有用信息。稍后将详细地展示这种效果。

人工神经元可以通过它们能够接收的输入的种类（二元值或连续值）和用来产生输出的转换的种类（激活函数）来定义。在 DL4J 中，同一层的所有神经元具有相同的激活函数。

连接权重。神经网络中的连接权重是对网络中指定神经元的输入信号进行缩放（放大或最小化）的系数。在神经网络中，这些一般表示为从一点到另一点的线或箭头，即数学图的边。在神经网络的数学表示方法中，连接权重通常表示为 w。

偏置。偏置是被加到输入的标量值，以确保无论信号强度如何，每层至少有几个节点被激活。偏置使得学习能够在低信号事件触发的网络中进行。它允许网络尝试新的解释或行为。偏置通常表示为 b，并且像权重一样，会在整个学习过程中被修改。

激活函数。控制人工神经元行为的函数称为激活函数。输入的传输被称为**前向传播**。激活函数对输入、权重和偏置整体进行转换。这些转换的结果作为下一个节点层的输入。在神经网络中使用的许多（但非全部）非线性转换将数据转换到一个方便处理的范围，例如 0 到 1，或 −1 到 1。当一个人工神经元传递一个非零值到另一个人工神经元时，它就被激活了。



激活值

激活值是每个从前一层传递到下一层的值。这些值是每个人工神经元的激活函数的输出。

2.3 节将介绍广义的神经网络中激活函数的不同类型及其一般功能。



激活函数及其重要性

本书之后几乎每一章都会持续讲解激活函数及其用法。DL4J 库使用围绕不同类型激活函数的、基于层的架构。

2. 生物神经元与人工神经元的比较

如果重新思考作为人工神经元基础的生物神经元，我们可能会问：“人工神经元与生物神经元是如何匹配的？”人工神经元的概念在生物神经元上的对应关系是：输入连接功能由生物神经元中的树突提供，计算总和的功能由神经元胞体提供。最后，激活功能由生物神经元中的轴突提供。



比较的局限性

我们（再次）注意到，生物神经元仍然比人工神经元更复杂。研究正在努力更好地理解生物神经元的功能。

3. 前馈神经网络架构

了解了人工神经元和感知器之间的区别，就可以更好地理解全连接多层前馈神经网络的架构。在多层前馈神经网络中，人工神经元被排列成组，这些组称为层。以层概念为基础，多层神经网络具有以下组成部分：

- 一个输入层；
- 一个或多个全连接的隐藏层；
- 一个输出层。

如图 2-7 所示，每一层的神经元（用圆圈表示）都完全连接到所有相邻层的所有神经元。

每层的神经元（大部分时间）都使用相同类型的激活函数。输入层的输入是原始的向量输入。其他层神经元的输入是前一层神经元的输出（激活值）。当数据以前馈的方式通过网络移动时，会受到连接权重和激活函数类型的影响。下面看看每一类层的细节。

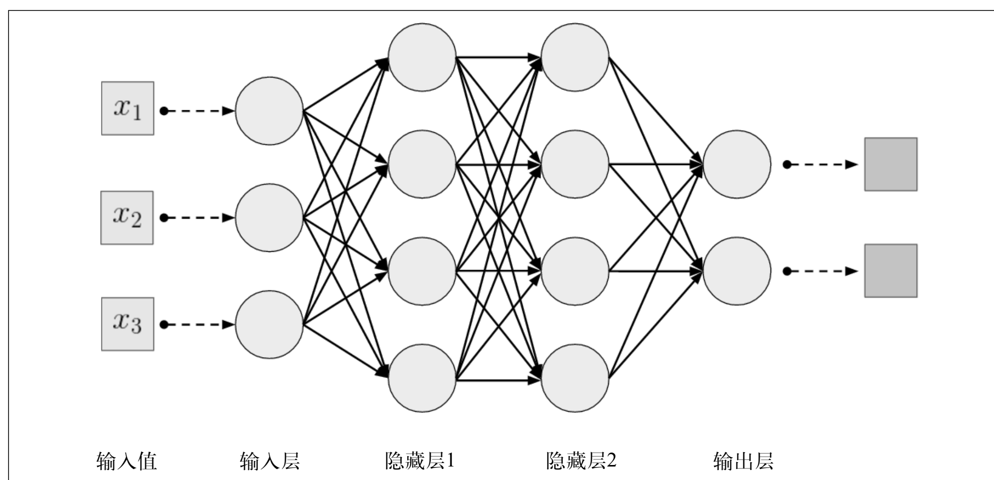


图 2-7：全连接多层前馈神经网络的拓扑

输入层。这一层是输入数据（向量）进入网络的入口。输入层中神经元的数量通常与网络输入特征的数量相同。输入层之后是一个或多个隐藏层（稍后解释）。典型前馈神经网络中的输入层完全连接到下一个隐藏层，但在其他网络架构中，输入层也许不是完全连接的。

隐藏层。前馈神经网络中有一个或多个隐藏层。层与层之间连接的权重值是神经网络对从原始训练数据中提取的学习信息进行编码的方法。隐藏层是神经网络对非线性函数建模的关键，正如在单层感知器网络的局限性中所看到的。

输出层。模型的输出层给出答案或预测。鉴于使用神经网络模型将输入空间映射到输出空间，输出层基于输入层的输入给出输出。根据神经网络的设置，最终输出是实值输出（回归）或一组概率（分类）。这由应用于输出层神经元上的激活函数的类型所控制。输出层通常使用 softmax 或 sigmoid 激活函数进行分类。本章稍后将探讨这两种激活函数之间的区别。

层之间的连接。在全连接的前馈网络中，层之间的连接是从前一层中的所有神经元到下一层中的所有神经元的外出连接。随着算法使用反向传播学习算法找到它所能找到的最佳解决方案，我们逐步改变这些权重。在数学上，可以将权重理解为 1.3 节中的参数向量，该节将机器学习描述为通过优化参数向量（例如，这里的“权重”）最小化误差的过程。

介绍过了前馈神经网络的基本结构，本章剩余部分将详细介绍反向传播的训练机制及激活函数的细节，最后会介绍常用的损失函数和超参数。

2.2 训练神经网络

一个训练有素的人工神经网络具有能够放大信号和抑制噪声的权重。更大的权重意味着信号和网络结果之间的联系更紧密。与权重值较小的输入相比，权重值较大的输入会对网络对数据的解释有更大影响。

使用权重训练任何学习算法是调整权重和偏置的过程，它使其中一些值更小，另外的值更大，从而将重要性分配给某些信息位并最小化其他的位。这有助于模型了解哪些预测因子（或特征）与哪些结果联系在一起，并相应地调整权重和偏置。

在大部分数据集中，某些特征与某些标签密切相关（例如，面积与房屋售价有关）。神经网络基于输入和权重进行猜测，然后评估结果的准确度，使用蛮力来学习这些关系。优化算法中的损失函数，如 SGD，会因好的猜测而奖励网络，因坏的猜测而惩罚网络。SGD 会移动网络参数——接近好的预测，远离坏的预测。

另一种看待学习过程的方法是将标签视为理论，将特征集视为证据。然后对网络寻求建立理论和证据之间的相关性做出类推。该模型试图回答“证据支持哪个理论”的问题。基于这些思想，接下来介绍与神经网络相关的最常用的学习算法：**反向传播学习**。

反向传播学习

反向传播是神经网络模型中减小误差的一个重要部分。为了解释反向传播，要回到对信息

如何在前馈神经网络中传播的讨论。先了解这种学习算法是如何工作的，然后再深入研究反向传播学习的数学符号和伪代码。



反向传播学习的起源

反向传播学习由 Bryson 和 Ho 于 1969 年发明。在神经网络训练的研究和实践中，它长期被忽略，直到 20 世纪 80 年代中期才复兴。

1. 算法的直觉

反向传播学习类似于感知器学习算法。我们希望随着网络中的前向传递计算输入样本的输出。如果输出匹配标签，就什么也不做；如果输出与标签不匹配，则需要调整神经网络中连接的权重。

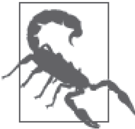
为了详细说明一般神经网络的学习过程，先来看看算法的伪代码，如示例 2-1 所示。

示例 2-1　一般的神经网络训练伪代码

```
function neural-network-learning( training-records ) returns network
    network <- initialize weights (随机)
    start loop
        for each example in training-records do
            network-output = neural-network-output( network, example )
            actual-output = 观测到的样本的实际输出
            update weights in network based on
                { example, network-output, actual-output }
        end for
    end loop when 所有样本都被正确预测或者达到了停止条件
    return network
```

关键要明确哪些权重产生了误差，在导致误差产生的权重之间划分责任。感知器学习算法要做到这一点很容易，因为每个输入只有一个权重会影响输出值。而对前馈多层神经网络学习算法来说，挑战则更大。网络中有很多权重，连接每个输入和输出，所以做到这一点更加困难。每个权重都影响多个输出，所以学习算法必须更聪明。

反向传播是一种为每个权重划分误差责任的实用方法。它类似于感知器学习算法。反向传播试图最小化与训练的输入相关联的标签的（或者“实际的”）输出和从网络输出产生的值之间的误差。稍后将介绍大部分神经网络文献中使用的前馈神经网络的反向传播的数学符号。



多层网络学习算法的一个注意事项

多层神经网络的学习算法既不能保证收敛到全局最优，也不是绝对高效的。它的一个直接影响是，在最坏的情况下，如何从训练样本得到通用的函数是一个棘手的问题。不过在一些适合的超参数调优中，学习算法在实践中表现得很好。

2. 反向传播的详细研究

本书的绝大部分无意展示大量数学公式，然而对于反向传播的主题，为了便于你理解作为本书大部分内容之基石的一个核心概念，我们有必要设置一节来用数学符号说明这些概念。



关于符号的说明

这种符号类似于你在机器学习的会议论文或者著名的机器学习教科书中所看到的。我们希望用一种让你感觉舒适的方式来解释符号。理想情况下，这是为你准备的一块跳板，此后你将能够探索更多的神经网络和深度学习的论文。

放大前一个示意图，聚焦在输入层和第一个隐藏层上，如图 2-8 所示。

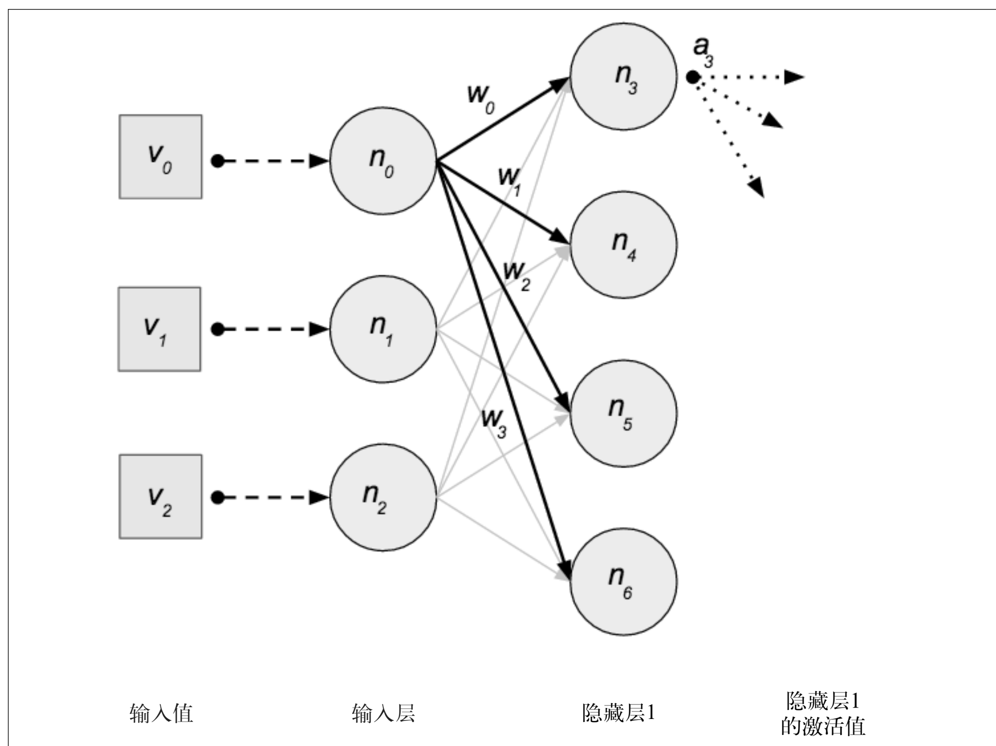


图 2-8：放大后的多层感知器网络，标上了组件的标签

图 2-8 中是之前的多层神经网络示意图放大后的样子，并且图中的一些符号也更新了。我们将用这些符号来解释反向传播。表 2-2 列出了图中的符号。

表2-2：神经网络的符号

符　　号	解　　释
i	人工神经元的索引
n_i	索引 i 处的神经元
j	前一层中连接神经元 i 的神经元索引
a_i	神经元 i 的激活值（神经元 i 的输出）
$\boldsymbol{A}_i$	神经元 i 的输入的激活值向量
g	激活函数
g'	激活函数的导数

（续）

符　号	解　释
Err_i	网络的输出与训练样本的实际输出值之间的差异
$\boldsymbol{W}_i$	指向神经元 i 的权重向量
$W_{j,i}$	从前一层的神经元 j 到神经元 i 的输入连接的权重
$input_sum_i$	神经元 i 的输入的加权和
$input_sum_j$	前一层中神经元 j 的输入的加权和（用于反向传播）
α	学习率
Δ_j	前一层连接神经元 j 的误差项
Δ_i	神经元 i 的误差项 ; $= Err_i \times g'(input_sum_i)$

为了详细解释该算法，先来看看反向传播学习算法的伪代码，如示例 2-2 所示。

示例 2-2　更新权重的反向传播算法伪代码

```
function backpropagation-algorithm
    ( network, training-records, learning-rate ) returns network
    network <- initialize weights (随机)
    start loop
        for each example in training-records do

            //为这个输入样本计算输出
            network-output <- neural-network-output( network, example )

            //计算输出层中神经元的误差和[增量]
            example_err <- target-output - network-output

            //更新通向输出层的权重
            W_{j,i}←W_{j,i}+α×a_j×Err_i×g′(input_sum_i)

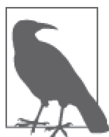
            for each subsequent-layer in network do

            //计算每个节点的误差
            Δ_j←g′(input_sum_j)∑_i W_{j,i}Δ_i

            //更新通向该层的权重
            W_{k,j}←W_{k,j}+α×a_k×Δ_j

        end for

    end for
end loop when network has converged
return network
```



关于伪代码中损失函数的说明

示例 2-2 的伪代码中没有显式调用损失函数（本章稍后介绍）。考虑到可读性，这里以算法的形式介绍了反向传播算法，因为这样对普通的实践者最友好。然而，考虑到数学思维较强的实践者，附录 C 从数学角度解释了反向传播。

在这种情况下，Err_i 项依赖损失函数的导数。我们使用均方误差（MSE），因此导数不同。

3. 理解反向传播的伪代码

示例 2-2 具有以下输入。

- 网络：一个多层前馈神经网络。
- 训练记录：一组具有相关输出的训练向量。
- 学习率：学习的速率（有时用希腊字母 α 表示）。

初始化神经网络以启动算法，并使用输入样本开始循环（直至遇到终止条件或达到最大轮数）。首先计算当前网络对当前输入样本的输出，比较该输出与输入相关的实际输出，并计算误差。

现在准备计算通向输出层的权重更新。

更新输出层权重。输出层权重更新通过以下方式计算：

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \mathrm{Err}_i \times g'(\mathrm{input_sum}_i)$$

这是神经网络中神经元 j 和神经元 i 之间的所有连接的权重更新规则。图 2-9 突出显示了这些通向输出层的连接。

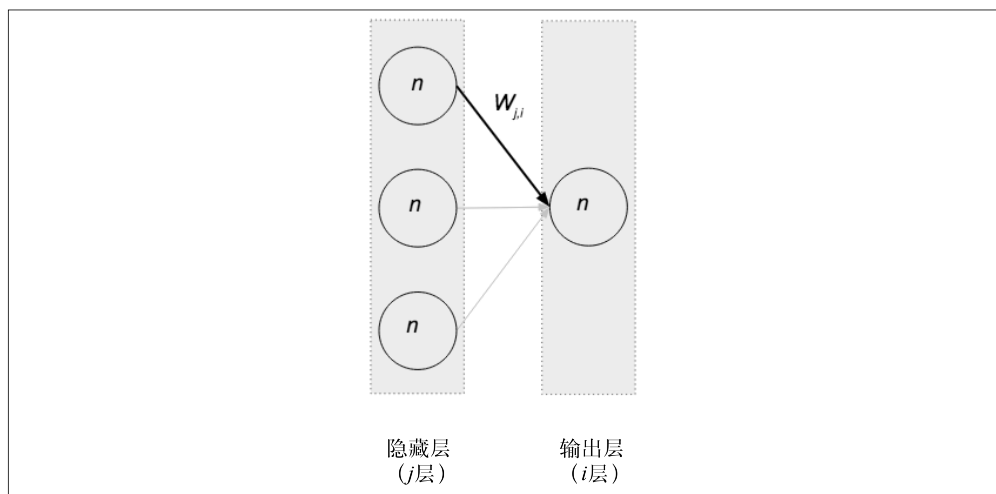


图 2-9：更新进入输出层的连接

仔细查看这个示意图，能看到正在研究的神经元 j 的权重从前一个隐藏层连接到当前神经元 i。我们将学习率 α（本章稍后探讨）乘以神经元 j 传入的激活值。这个输入通过获得神经元 j 的净输入，然后计算神经元 j 的激活值算出。

为了计算输入神经元 i 的激活函数的总权重，要计算输入权重 $\boldsymbol{W}_j$ 和激活向量 $\boldsymbol{A}_j$ 的点积，然后添加偏置项的值：

$$\mathrm{input_sum}_i = \boldsymbol{W}_i \cdot \boldsymbol{A}_i + b$$

神经元 j 激活值的计算公式如下所示：

$$a_j = g(\mathrm{input_sum}_j)$$

神经元 i 上样本 e 的误差项表示为 Err_i。激活函数的导数表示为 $g'(x)$，它被应用于神经元 i 的净输入，表达式如下所示：

$$g'(\text{input_sum}_i)$$

这个更新规则类似于更新感知器的方法，区别在于使用前一层的激活值，而非其原始输入值。该规则还包含激活函数导数的项，以得到激活函数的梯度。

进一步表示误差项。前面介绍了激活函数导数的表达式为 $g'(z)$。误差项通常表示为 Δ_i，它的表达式如下所示：

$$\Delta_i = Err_i \times g'(\text{input_sum}_i)$$

这样就得到一个更紧凑的权重更新函数，如下所示：

$$W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$$

我们在现在探讨的这个伪代码的内部循环中使用了这个更新表达式。



权重空间中的梯度下降

我们把反向传播看作权重空间中的梯度下降，权重空间中的梯度是误差表面的梯度。这个误差表面将输入特征的误差描述为神经网络中权重值的函数。

误差值的新传播规则。误差变化值的传播规则现在变成下面的表达式：

$$\Delta_j \leftarrow g'(\text{input_sum}_j) \sum_i W_{j,i} \Delta_j$$

这就给出了输入层和隐藏层之间权重的一个新的更新规则。

更新隐藏层。利用反向传播算法，回过头来遍历隐藏层，更新每层之间的连接，直至到达输入层为止。图 2-10 突出显示了两个隐藏层之间的连接。

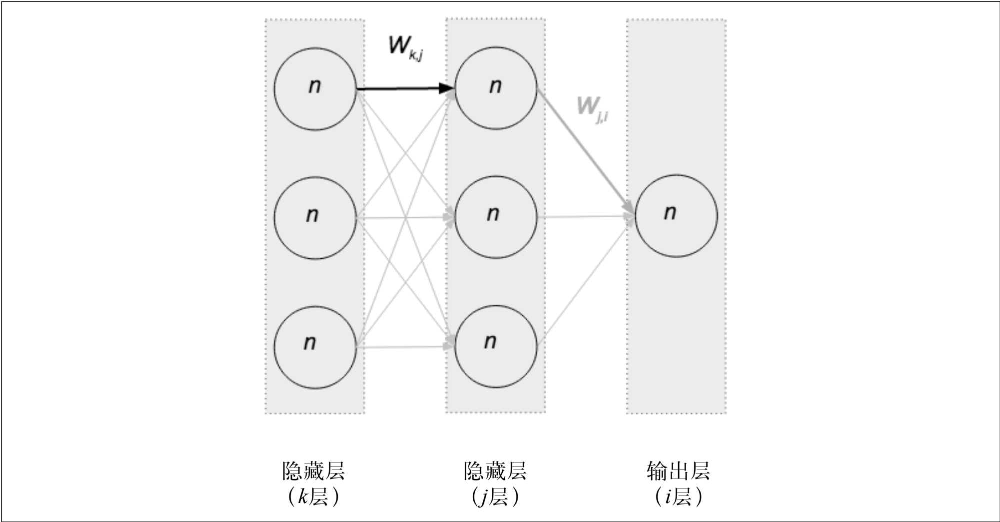


图 2-10：突出显示隐藏层之间的连接

为了更新这些连接，要从之前计算的分配后的误差值中获取输入，并将其与来自前一层连接的激活值（输入）和学习率相乘。

然后将其加到先前的权重值，以下是更新后的值：

$$W_{k,j} \leftarrow W_{k,j} + \alpha \times a_k \times \Delta_j$$

那些因产生误差而应承担责任的权重和偏置被削弱了，以限制其信号。而那些传递了支持正确答案的信号的权重和偏置得到了加强。将权重调整到最佳状态是一个迭代过程，需要一步一步地进行。

反向传播与误差责任的分配

反向传播通过网络反向分配“责任”。每一个向当前节点发送输入的隐藏节点，对它前向连接的每个神经元的误差负有部分责任。

第一个隐藏层使用来自原始特征向量的输入作为输入，所有随后的层使用前一层神经元的激活值作为输入。但是对于输出层之前的隐藏层，必须适当分配误差。通过误差反向传播，根据隐藏节点与输出节点之间的连接权重划分 Δ_i 值。

使用以下方程计算前面方程中的 Δ_j：

$$\Delta_j = g'(\text{input_sum}_j)\sum_i W_{j,i}\Delta_i$$

该方程考虑了当前层中的每个节点 i，将当前误差值 Δ_i 乘以进入连接的权重和激活函数的导数。这产生了前一层中节点相应的误差值，我们用这个值更新进入该层的连接。逐层执行这个算法，直到网络的每一层都更新了。

这些学习步骤的长度，或者说每次迭代改变的权重的量，被称为**学习率**。学习率是定义的参数（不是对网络性能的评估）。本章稍后详细探讨超参数时将讨论学习率。

反向传播与小批量 SGD

第 1 章介绍了一种被称为小批量的 SGD 变体，该方法一次训练多个样本，而非一次只训练一个样本。神经网络中的反向传播和 SGD 也使用小批量来改进训练。

在内部，我们计算小批量中所有样本的平均梯度。具体说来，通过一系列线性代数矩阵运算，计算所有样本的前向传递以得到其输出分数。在每层的反向传递中，（为该层）计算梯度的平均值。通过这样的反向传播，能够得到更好的梯度近似，同时能更高效地使用硬件。



人工智能的寒冬 II：20 世纪 90 年代初

20 世纪 80 年代末及 90 年代初，专家系统和 LISP 机器等技术被过度推广，这两种技术都未能达到预期。美国国家战略计算计划在这个周期结束时取消了新的支出预算，第五代计算机的目标未能实现。

2.3 激活函数

使用激活函数将一层节点的输出传递到下一层（直至进入输出层）。激活函数是从标量到标量的函数，产生神经元的激活值。我们使用神经网络中隐藏神经元的激活函数，将非线性引入网络的建模能力中。许多激活函数属于逻辑类别的转换，（当图形化时）看起来像字母 S，这类函数被称为 S 形函数。S 型函数家族包含多个变体，其中一个就是著名的 sigmoid 函数。下面介绍神经网络中一些有用的激活函数。

2.3.1 线性函数

线性转换（见图 2-11）基本上是恒等函数，$f(x) = Wx$，其中因变量与自变量具有直接的比例关系。实际上这意味着函数传递着不变的信号。

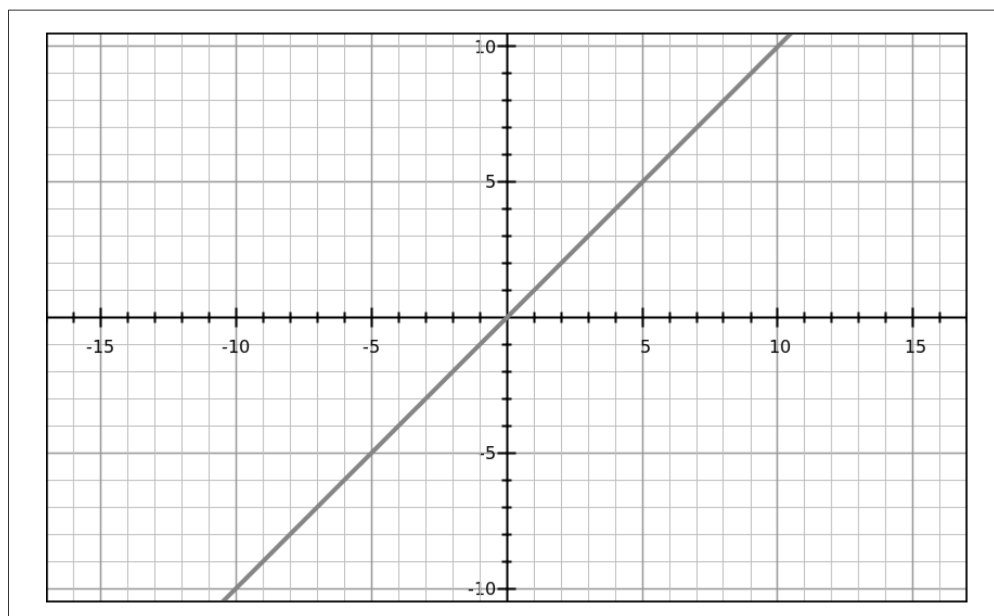


图 2-11：线性激活函数

这种激活函数用于神经网络的输入层中。

2.3.2 sigmoid函数

和所有逻辑转换一样，sigmoid 函数可以减少数据中的极值或离群值，而不必将其移除。图 2-12 中的垂直线是决策边界。

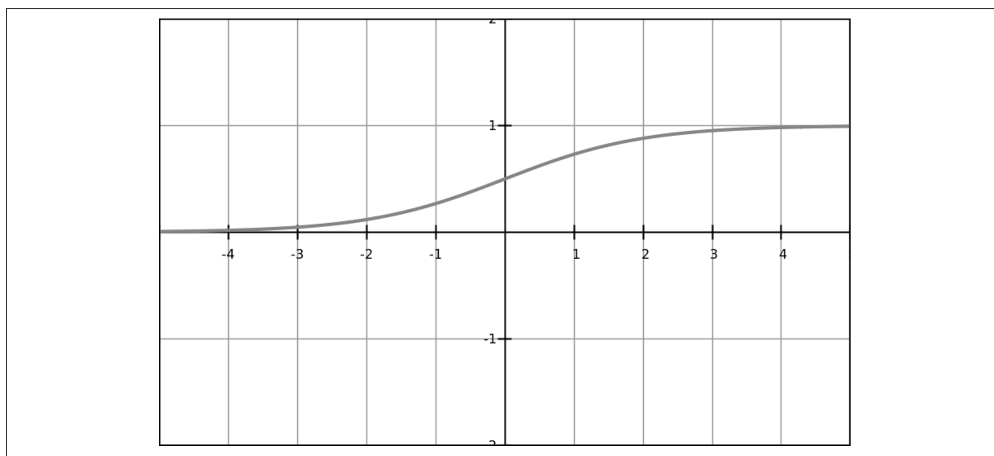


图 2-12：sigmoid 激活函数

sigmoid 函数是一种把几近无限范围的自变量转换成 0 和 1 之间简单概率的函数，它的大部分输出非常接近 0 或 1。



理解 sigmoid 函数的输出

sigmoid 激活函数为每个分类输出一个独立的概率。

2.3.3 tanh函数

tanh 的发音为 tanch，是双曲三角函数（见图 2-13）。正如正切表示直角三角形对边和邻边之间的比率，tanh 表示双曲正弦与双曲余弦之比：tanh(x)=sinh(x)/cosh(x)。与 sigmoid 函数不同，tanh 函数的规范化范围为 -1 到 1，其优势在于可以更容易地处理负数。

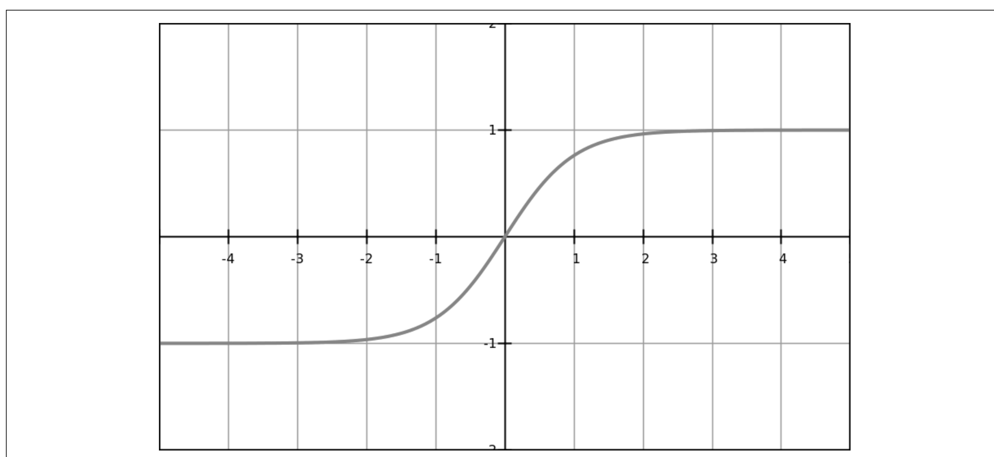


图 2-13：tanh 激活函数

2.3.4 hard tanh函数

与 tanh 函数类似，hard tanh 函数简单地强制规定了规范化范围，任何超过 1 的数值都被转化为 1，任何小于 -1 的值都被转化为 -1。它适合作为更简单直接的激活函数，允许有限的决策边界。

2.3.5 softmax函数

softmax 是逻辑回归的一个泛化，因为它可以应用于连续数据（而不是二元分类），并且可以包含多个决策边界。它处理多项式标签系统。在分类器的输出层中经常能看到 softmax 函数。



理解 softmax 函数的输出

softmax 激活函数返回在互斥的输出类别上的概率分布。

为了详细说明 softmax 输出层的概念以及使用方法，考虑两个用例。如果有一个多分类建模问题，但我们只关心这些类别的最高得分，我们将使用一个带有 `argmax()` 函数的 softmax 输出层来获得所有类别的最高得分。



多分类处理

如果想得到每个输出的多个分类（例如，“人 + 车”），不应使用 softmax 作为输出层。相反，应使用 sigmoid 输出层分别给出每个类别的概率。

对于那些有一大组标签（例如，数以千计的标签）的情况，要使用 softmax 激活函数的变体，它被称为**分级 softmax**（hierarchical softmax）激活函数。该变体将标签分解为树形结构，并在树的每个节点训练 softmax 分类器来指导分支进行分类。

2.3.6 修正线性函数

修正线性（rectified linear）是一种更有趣的转换，仅当输入高于某个数量时，它才激活节点。当输入小于零时，输出为零，但当输入上升到某一阈值以上时，它与因变量产生线性关系 $f(x)=\max(0, x)$，如图 2-14 所示。

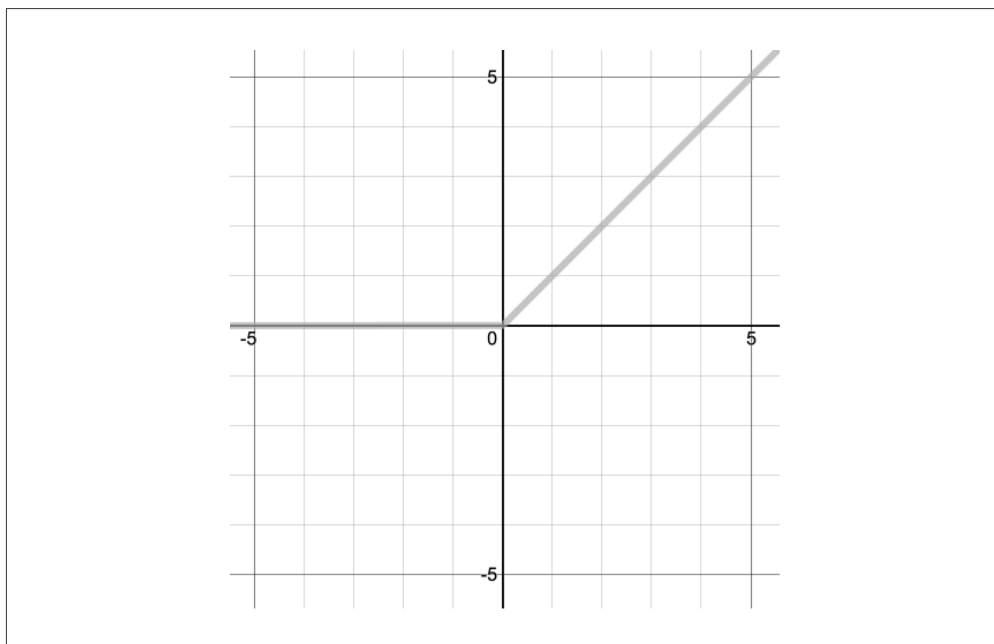


图 2-14：修正线性激活函数

修正线性单元（ReLU）代表了当前的技术发展水平，它已被证明可以在许多不同的情况下工作。因为 ReLU 的梯度是常数（可为 0），所以它可以解决梯度消失和梯度爆炸问题。实践证明 ReLU 激活函数的训练效果好于 sigmoid 激活函数。



ReLU 激活函数不可思议的有效性

与 sigmoid 和 tanh 激活函数相比，ReLU 激活函数不受梯度消失问题的影响。使用 hard tanh 作为激活函数会导致该层的激活输出中出现稀疏性。研究表明，使用 ReLU 激活函数的深度网络可以训练得很好，而无须使用预训练技术。

1. Leaky ReLU

Leaky ReLU 是一种缓解“死亡 ReLU”问题的策略。当 $x < 0$ 时，与 ReLU 令函数值为零的做法不同，Leaky ReLU 会设置一个小的负斜率（例如“0.01 左右”）。实践中已经有了一些成功的 ReLU 变体，但结果不太稳定。方程如下所示：

$$f(x) = \begin{cases} x & x > 0 \\ 0.01x & x \leqslant 0 \end{cases}$$

2. softplus

该激活函数被视作“ReLU 的平滑版本”，如图 2-15 所示。比较此图与图 2-14 中的 ReLU 的图形。

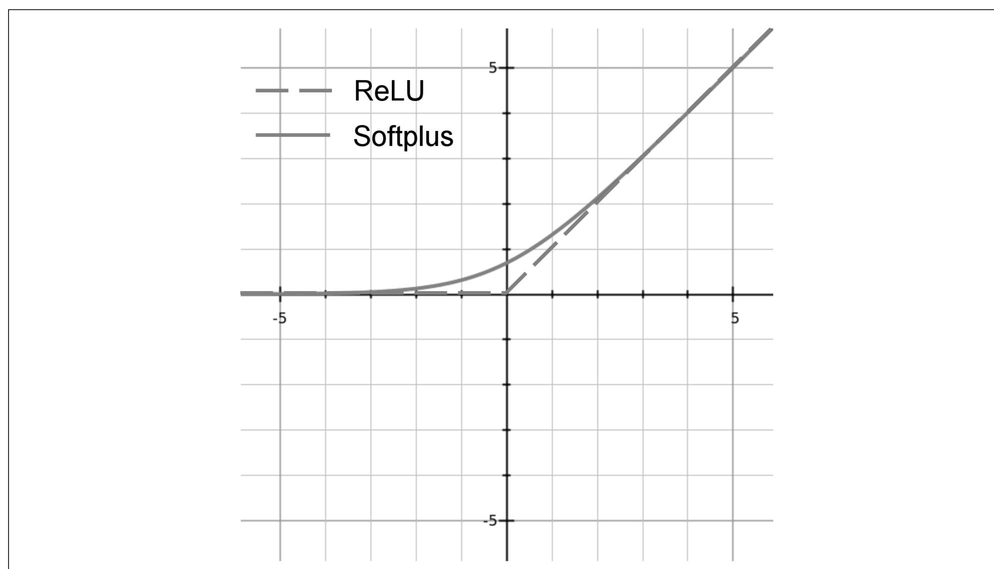


图 2-15：ReLU 和 softplus 激活函数的图形

图 2-15 中的 softplus 激活函数（$f(x) = \ln[1 + \exp(x)]$）的形状与 ReLU 相似。图中 softplus 处处可微且导数值非零，这一点与 ReLU 不同。

2.4　损失函数

损失函数量化了给定神经网络的训练结果与理想值的接近程度。这个思路很简单，根据在网络预测中观察到的误差来评估，然后将整个数据集的误差汇总起来，求它们的平均值，这样就有了能够代表神经网络与理想值接近程度的单个数值。

寻找这种理想状态相当于找到一组参数（权重和偏置），这些参数使由误差引起的“损失”最小。通过这种方式，损失函数有助于将神经网络的训练问题转化为优化问题。在大多数情况下，这些参数不能用解析方法求解，但通常可以通过梯度下降等迭代式优化算法来逼近。下面简要介绍常见的损失函数，并在必要时介绍它们在机器学习中的起源。

2.4.1　损失函数的符号

本节的方程将使用这里所描述的符号。

- 考虑收集来用于训练神经网络的数据集。N 表示已收集的样本（具有相应的结果的输入集）数量。
- 考虑所收集的输入和输出的性质。每个数据点记录一些独特的输入特征和输出特征。P 表示收集的输入特征的数量，M 表示已观察到的输出特征的数量。
- (X, Y) 表示收集的输入和输出数据。注意，将有 N 个这样的对，其中输入是 P 个值的集合，输出 Y 是 M 个值的集合。数据集中的第 i 对表示为 X_i 和 Y_i。

- $\hat{Y}$ 表示神经网络的输出。由于 $\hat{Y}$ 是网络对 Y 的猜测，因此它也具有 M 个特征。
- 符号 $h(X_i) = \hat{Y}_i$ 表示神经网络转换输入 X_i 以给出输出 $\hat{Y}_i$。稍后会修改这个符号，以强调它对权重和偏置的依赖。
- 当引用第 j 个输出特征时，用它作为下标，将符号正确链接到一个矩阵，其中行是不同的数据点，而列是不同的独特特征。因此，$Y_{i,j}$ 指在所收集的第 i 个样本中观察到的第 j 个特征。
- $L(W, b)$ 表示损失函数。

对于指定的可用数据，损失函数的符号表示其值仅依赖 W 和 b，即神经网络的权重和偏置，这一点非常重要。在一个给定的神经网络世界中，有一系列的层、配置等，它们都基于给定的一组数据训练，损失函数的值完全取决于网络的状态，即权重和偏置。这些值发生变化，则损失也发生变化。指定的输入发生变化，则输出也将发生变化。

因此，符号 $h(X) = \hat{Y}$ 以一组权重和偏置为条件，所以符号修订为 $h_{w,b}(X) = \hat{Y}$。现在准备好处理损失函数了。

2.4.2　用于回归的损失函数

本节讨论适合于回归模型的损失函数。

1. 均方误差损失

处理需要实值输出的回归模型时，使用平方损失函数，这很像线性回归中普通的最小二乘法的情况。考虑一下只能预测一个输出特征（$M = 1$）的情况。对预测误差取平方，然后用数据点的数量取平均值，朴实又简单，这样的 MSE 损失方程如下所示：

$$L(W,b) = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

如果 $M > 1$，并且我们期望预测给定输入特征集的多个输出特征，该如何处理呢？在这种情况下，期望值和预测值，Y 和 $\hat{Y}$，分别是一个有序的数字列表，即向量。



关于损失函数的说明

损失函数将期望和预测之间的差异由向量转化为一个数值。

下面是 MSE 损失函数的另一个变体：

$$L(W,b) = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{M}\sum_{j=1}^{M}(\widehat{y_{i,j}} - y_{i,j})^2$$

如果你熟悉线性代数，会认出上面方程中的内层 Σ 是对欧几里得距离的平方求和。实际上，MSE 有时被这些术语提及。请注意数据集的大小 N 和网络需要预测的特征数量 M 都是常数。因此它们是简单的缩放因子，可以用其他方式来解释（比如通过缩放学习率）。在许多使用场景下（包括 DL4J），M 被丢弃，并且为了方便数学计算，在表达式中增加除以 2 的部分（在反向传播梯度的计算中，这将变得更清晰）。下面的等式是 DL4J 库中用于

回归的 MSE 版本：

$$L(W,b)=\frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\widehat{y_{i,j}}-y_{i,j})^2$$



MSE 真的是一个凸损失函数吗？

在技术意义上，MSE 是凸损失函数，然而，当处理神经网络中的隐藏层时，凸特性不再成立，因为有多个参数集能得到相同的损失值。



优化 MSE

优化 MSE 等价于优化平均值。

2. 用于回归的其他损失函数

虽然 MSE 被广泛使用，但它对离群值非常敏感，在选择损失函数时需要考虑这一点。当我们选择某支股票进行投资时，会把离群值考虑在内，但在购房时，却不会考虑。大多数人为最感兴趣的东西付钱，在这种情况下，我们对中位数更感兴趣，而对平均值不太感兴趣。

平均绝对误差损失。在相似的场景中，平均绝对误差（MAE）损失能够代替 MSE 损失，如下式所示：

$$L(W,b)=\frac{1}{2N}\sum_{i=1}^{N}\sum_{j=1}^{M}\left|\widehat{y_{i,j}}-y_{i,j}\right|$$

它简单地在整个数据集上计算绝对误差的平均值。

均方对数误差损失。另一个用于回归的损失函数是均方对数误差（MSLE）：

$$L(W,b)=\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}(\log\widehat{y_{i,j}}-\log y_{i,j})^2$$

平均绝对百分比误差损失。最后还有平均绝对百分比误差（MAPE）损失：

$$L(W,b)=\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M}\frac{100\times\left|\widehat{y_{i,j}}-y_{i,j}\right|}{y_{i,j}}$$

3. 对回归损失函数的讨论

这些损失函数都是有效的选择，当然没有任何一个损失函数在所有场景中都优于其他的损失函数。MSE 的使用非常广泛，在大多数情况下是安全的选择，MAE 也是同样。如果网络正在预测在很大范围内变化的输出，那么 MSLE 和 MAPE 值得考虑。假设用一个网络来预测两个输出变量：一个在 [0, 10] 范围内，另一个在 [0, 100] 范围内。在这个场景中，相比第一个输出，MAE 和 MSE 将更明显地惩罚第二个输出的误差。MAPE 使其成为相对误差，因此不基于范围来区分。MSLE 将所有输出的范围缩小，简单地将 10 和 100 转换为 1 和 2（以 10 为底的对数）。



神经网络的回归中使用的一般实践

虽然 MSLE 和 MAPE 是处理大范围数据的方法，但神经网络的一般做法是将输入规范化至合适的范围，并使用 MSE 或 MAE 优化平均值或中值。

2.4.3　用于分类的损失函数

我们可以建立神经网络把数据点分成不同的类别，例如欺诈 | 不是欺诈。然而在为分类问题构建神经网络时，重点往往是将概率附加到这些分类（30% 欺诈 | 70% 不是欺诈）上。不同的场景需要不同的损失函数。

1. hinge损失

当需要优化网络以进行硬分类时，最常用的是 hinge 损失函数。例如，0 = 无欺诈和 1 = 欺诈，通常称之为 0–1 分类器。0 和 1 的选择也许有点武断，–1 和 1 也可以代替 0–1。也可以将 hinge 损失用于一类名为“最大间隔分类”的模型（例如支持向量机，一个与神经网络关系有点疏远的“表亲”）。

当数据点必须被分类为 –1 或 1 时，hinge 损失方程如下所示：

$$L(W,b)=\frac{1}{N}\sum_{i=1}^{N}\max(0,1-y_{i,j}\times\widehat{y_{i,j}})$$

hinge 损失主要用于二元分类。也有扩展到多分类（例如，“一对所有”“一对一”）的 hinge 损失，这里没有涉及。



hinge 损失是一个凸函数

注意，hinge 损失与 MSE 一样，都是凸函数。

2. 逻辑损失

逻辑损失函数用于对概率比对硬分类更感兴趣的场景。例如，用人工循环的解决方案来标记潜在的欺诈行为，以及涉及费用支出的对“用户点击广告的概率”的预测。

预测有效的概率意味着生成 0 和 1 之间的数字，也意味着确保互斥结果的概率总和等于 1。出于这个原因，在神经网络的最后一层分类中使用 softmax 至关重要。需要注意的是，sigmoid 激活函数也会给出 0 和 1 之间的有效值。但在输出互斥的场景下不能使用它，因为它不对输出值之间的依赖关系建模。

既然已经确信神经网络会为现有的类别生成有效的概率，我们就可以将精力集中在损失函数和需要优化的地方。我们希望优化所谓的“最大似然”，换句话说，我们希望最大化预测正确类别的概率，并且对每个样本都这样做。

考虑一下网络预测两个类别的概率的情况，例如欺诈和非欺诈的 0–1 分类器。基于前面描述的符号，可以将给定输入 X_i 的输出表示为 $h(X_i)$ 和 $1-h(X_i)$，将一组权重和偏置表示为 W 和 b，那么 1 和 0 的概率的表达式如下所示，分别为：

$$P(y_i = 1 \mid X_i; \boldsymbol{W}, \boldsymbol{b}) = h_{\boldsymbol{W},\boldsymbol{b}}(X_i)$$
$$P(y_i = 0 \mid X_i; \boldsymbol{W}, \boldsymbol{b}) = 1 - h_{\boldsymbol{W},\boldsymbol{b}}(X_i)$$

把这些方程组合起来，表示如下：

$$P(y_i \mid X_i; \boldsymbol{W}, \boldsymbol{b}) = (h_{\boldsymbol{W},\boldsymbol{b}}(X_i))^{y_i} \times (1 - h_{\boldsymbol{W},\boldsymbol{b}}(X_i))^{1-y_i}$$

在概率的语境下，最大似然的口头定义中的“并且”这个词应该立即引起注意。在所有可用样本中“并且”要转化为所讨论的概率的乘积问题，如下所示：

$$L(W,b) = \prod_{i=1}^{N} \hat{y}_i^{\,y_i} \times (1 - \hat{y}_i)^{1-y_i}$$

下面介绍负对数似然。

负对数似然。为了便于数学计算，当处理概率的乘积时，习惯上将它们转换成概率的对数，因此概率的乘积将转化为概率的对数之和。



负对数似然与概率最大化

对数是一个单调递增函数，因此最小化负对数似然等价于概率最大化。

表达式还加了负号，使得方程现在相当于一个“损失”。这样，损失函数就变成了下面的形式，通常称之为负对数似然：

$$L(W,b) = -\sum_{i=1}^{N} y_i \times \log \hat{y}_i + (1 - y_i) \times \log(1 - \hat{y}_i)$$

将损失函数从两个类别扩展到 M 个类别，就有了如下的逻辑损失方程：

$$L(W,b) = -\sum_{i=1}^{N} \sum_{j=1}^{M} y_{i,j} \times \log \widehat{y_{i,j}}$$

这在数学上相当于两个概率分布之间的交叉熵——在这种情况下，预测的和观察到的基于同样的标准，后面的章节将深入介绍这一点。



关于损失函数的统一观点

注意，最小化损失函数是通过最大化似然来实现的，而最大化似然又是通过最小化负对数似然来实现的。有些拗口？是的，确实如此。

交叉熵源于信息论，而用于分类的负对数似然法来自统计建模。这两种方法在数学上是相同的，所以不管用哪种方法，都可能让人困惑。

2.4.4 用于重建的损失函数

这组损失函数与所谓的**重建**有关。其背后的思想很简单，就是训练神经网络以尽可能地重建它的输入。那么这与记忆整个数据集有何不同？这里的关键是调整场景，使得网络被迫学习数据集之间的共性和特征。

一种方法是限制网络中的参数数量，使得网络被迫压缩数据，然后重建数据。另一种常用的方法是用无意义的“噪声”破坏输入，然后训练网络来忽略噪声并学习数据。这类神经网络的例子包括受限玻尔兹曼机、自动编码器等，它们都使用源于信息论的损失函数。

以下是 KL 散度的方程：

$$D_{KL}(Y \| \hat{Y}) = -\sum_{i=1}^{N} Y_i \times \log\left(\frac{Y_i}{\hat{Y}_i}\right)$$

虽然刚才简要探讨了交叉熵，并用概率的对数将和转化为乘积，但没有提到的是，概率的对数将我们直接带到了信息论领域，接触到了熵的概念。



不同的方法

尽管如上所述，负对数似然在数学上等同于交叉熵，但它们以不同的理论方法为基础。

2.5 超参数

在机器学习中，既有模型的参数，又有通过调优来使网络训练得更好更快的参数。这些调优参数被称为**超参数**，它们负责在学习算法训练期间控制优化函数及模型的选择。DL4J 中还将优化算法称作更新器，因为更新与算法在权重空间上最小化误差的步骤同义。

选择超参数的重点是确保模型既不欠拟合也不过拟合训练数据集，同时尽可能快地学习数据结构。

2.5.1 学习率

学习率影响为了最小化神经网络猜测的误差而进行优化的过程中参数调整的量。在神经网络穿越损失函数空间时，学习率作为系数，缩放了神经网络对其参数向量 $\boldsymbol{x}$ 所使用的步（更新）的大小。

在反向传播过程中将误差的梯度与学习率相乘，然后用这个乘积更新连接权重上一次迭代的值，以达到新的权重。学习率决定算法的下一次迭代使用多少梯度。较大的误差和陡峭的梯度与学习率相结合，会产生较大的步。当我们逼近最小误差且梯度变平时，步趋向于变短。

大的学习率系数（例如 1）会使参数出现跳跃，而小的学习率系数（例如，0.00001）会使参数变化缓慢。大的跳跃可以节省时间，但如果它们导致参数越过了最小值，结果将是灾难性的。学习率太大会使参数越过最低点，导致算法在最小值两侧来回跳动，无法停歇。

与之相反，小的学习率最终产生的误差值最小（它可能是局部最小值而不是全局最小值），但它可能导致训练时间变长，并且增加本已密集的计算进程的负担。神经网络在大数据集上的训练可能需要几周的时间，这时时间就非常宝贵了。如果你无法忍受一周后出结果，那就选择一个中等的学习率（例如 0.1），并与同一团队里的其他人一起实验，以获得最佳的速度和准确度。除了设置静态学习率之外，本书后面还将介绍如何随着时间的推移改变

学习率，以达到两全其美的效果。

2.5.2 正则化

正则化使用不同的方法，随着时间推移最小化参数大小，以帮助控制参数失控的影响。



在机器学习中抑制过拟合

正则化的主要目的是抑制机器学习中的过拟合。

在数学上，正则化用 λ 系数表示，用于在找到合适的拟合以及（随着特征的指数增长）维持某些较低的特征权重值之间折衷。

正则化系数 L1 和 L2 通过使某些权重更小，帮助解决过拟合。更小的权重值带来更简单的假设，而更简单的假设最为通用。特征集合中一些高阶多项式的未正则化的权重往往会使训练集过拟合。

随着输入训练集增大，正则化的影响将减小，参数趋向于增加。这是合适的，因为相对于训练集样本，过量的特征才是导致过拟合的首要元凶。更大的数据量才是终极的正则化。

2.5.3 动量

动量有助于学习算法摆脱搜索空间，否则学习可能陷入僵局。在更新器不知所措时，它帮助更新器找到通往最小值的路径。动量针对学习率，而学习率针对权重，动量有助于建立质量更高的模型。之后的许多章节将介绍动量超参数的作用。

2.5.4 稀疏

稀疏超参数让我们认识到，对于一些输入，只有几个特征是相关的。例如，假设网络可以对 100 万张图片分类。这些图片中任何一张都能通过有限数量的特征被识别出来。但为了有效地对数百万张图像进行分类，网络必须能够识别更多的特征，其中许多特征大部分时间并不出现。例如，海胆的照片不含鼻子和蹄子。对比来看，潜艇图片中鼻子和蹄子的特征也是 0。

海胆的特征在深广的神经网络层中少之又少，这是一个问题，因为稀疏的特征会限制激活节点的数量，阻碍网络的学习能力。为了解决稀疏性，偏置会迫使神经元激活，并使激活保持在平均值周围，以防止网络陷入停滞。

第3章

深度网络基础

你必须尽力地不停奔跑，才能使自己保持在原地。如果你想到别的地方，你奔跑的速度必须至少是现在的两倍！

——红桃王后，《爱丽丝镜中奇遇记》

3.1 定义深度学习

第 2 章介绍了机器学习和神经网络的基础知识，本章在此基础上介绍深度网络的核心概念，第 4 章将讲述具体的深度网络架构，然后第 5 章会介绍实际的例子。本章内容有助于你更好地理解后面的内容。首先回顾一下深度学习和深度网络的定义。

3.1.1 什么是深度学习

回顾第 1 章对深度学习的定义，其中一般的深度学习网络与典型前馈多层网络的区别如下所示：

- 比之前的网络拥有更多的神经元；
- 连接层的方式更复杂；
- “寒武纪大爆发”带来更强大的计算能力来进行训练；
- 自动特征提取。

说到“更多的神经元”时，我们的意思是为了表达更复杂的模型，神经元数量在过去几年中不断增长。层也从多层网络中每一层的全连接，进化到 CNN 中层之间神经元的局部连接，以及 RNN 中到同一神经元的循环连接（除了来自前一层的连接外）。

更多的连接意味着网络要优化更多的参数，这就需要在过去 20 年中发生的计算能力的爆发。所有这些进展为构建能够以更智能的方式提取特征的下一代神经网络打下了基础，使

得深度网络能够对较之前更复杂的问题空间建模（比如图像识别的进展）。随着行业需求的不断变化和扩展，神经网络的能力不得不提升。红桃王后[1]没有其他路可走。

1. 定义深度网络

为了进一步丰富深度学习的定义，首先定义深度网络的四个主要架构：

- UPN
- CNN
- RNN
- 递归神经网络

神经网络领域的研究还在继续，但是基于本书主题，我们将重点介绍这四种架构。在过去的 20 年中，这些架构已经发展得较为成熟。首先简单介绍这些技术的一些亮点，接着第 2 章开始的历史回顾，继续了解前馈多层神经网络的历史。

深度强化学习

Sutton 的 *Reinforcement Learning: An Introduction* 一书将**强化学习**定义为：

> 强化学习不是通过学习方法来定义的，而是通过学习问题来定义的。

可以说，任何适用于解决这类问题的方法都可以看作强化学习方法。强化学习不会告诉学习者采取什么行动，而是让代理通过模拟试验发现获得最佳奖励的行动。

在强化学习中，代理从没有经过训练的环境模型开始，且效用函数与代理追逐的奖励或目标同义。训练系统向代理提供来自环境的输入，并当模拟（或**游戏**）的（一个循环或**帧**的）结果为积极结果时，奖励代理。很多时候，行动不仅会影响当前的奖励，还会影响未来的奖励。试错和延迟奖励机制是强化学习的核心特征。

深度强化学习是强化学习的一个变体，其中神经网络被用作通用函数逼近器。这种方法的缺点是神经网络的行为不是有界的，并且收敛的证明不再成立。尽管如此，以神经网络作为通用函数逼近器仍能得到良好的结果。

2013 年，DeepMind 团队在 NIPS 2013 的深度学习研讨会上发表了一篇关于使用深度 Q 学习玩 Atari 游戏的论文。在这篇论文中，作者们使用了一个标准算法（Q 学习和函数逼近），该算法的函数逼近器就是一个 CNN。他们展示了一个能够玩 Atari 2600 游戏的代理，它以屏幕上的像素为输入，以 CNN 为内部模型。

当游戏基于所执行动作的结果为积极结果时，计算机 / 代理玩 Atari 游戏会得到肯定的奖励。对于一些游戏，该算法的游戏水平能够超过人类。

深度强化学习在本书写作过程中越来越受欢迎，希望本书将来的版本会涵盖它。本书附录 B 有一个介绍它的例子。

注 1：这里指刘易斯 • 卡罗尔的著作《爱丽丝镜中奇遇记》中的人物红桃王后，她必须不停奔跑，才能使自己保持在原地。

2. 进化过程与复苏

第 2 章提到过，神经网络在 20 世纪 80 年代中期进入了一个“寒冬”，那时人工智能所承诺的与它所能提供的不相符。正如历史上多次发生的那样，当神经网络这个有前景的技术跌落到泡沫化的底谷时（见图 3-1），许多研究者仍在为这个领域做着重要工作。

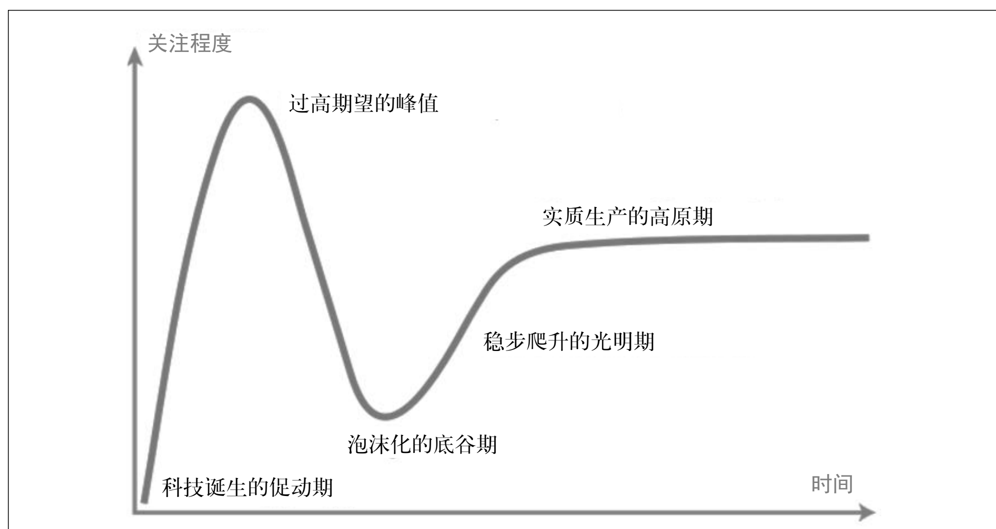


图 3-1：泡沫化的底谷

神经网络的一个重要发展是 Yann LeCun 在 AT&T 贝尔实验室开展的光学字符识别方面的工作。他的实验室专注于为金融服务部门做图像识别。通过这项工作，LeCun 和他的团队提出了图像识别的生物启发模型的概念，也就是今天大名鼎鼎的 CNN。这最终导致了 MNIST 手写基准的创建（本章稍后详细说明），并且通过深度学习实现了记录准确度的不断提升。



更好的标记数据

深度网络取得发展和成功的另一个促进因素是创建了更好、更大的标记数据集，如 MNIST 和 ImageNet。

20 世纪 80 年代末和 90 年代初，Sepp Hochreiter 等研究者利用 RNN 对时序数据建模的研究取得了进展。随着时间的推移，在 20 世纪 90 年代末，研究社区创造了更好的人工神经元变体，例如，长短期记忆（LSTM）单元以及带有遗忘门的记忆单元。神经网络正在世界各地的研究实验室里悄然复苏。

2000 年之后，研究人员和行业应用开发者开始逐步将这些进展应用于以下产品：

- 自动驾驶汽车
- 谷歌翻译
- 亚马逊 Echo 智能音箱
- AlphaGo

2006 年 Darpa 超级挑战赛中的自动驾驶汽车使用了许多技术，而不仅仅是深度学习。斯坦福大学和卡内基梅隆大学的顶级团队利用了图像处理的巨大改进。



计算机视觉研究的进展

2012 年，Alex Krizhevsky、IlyaSutskever 和 Geoffrey Hinton 开发了一个"大型深度卷积神经网络"，该模型赢得了 2012 年的 ILSVRC（ImageNet 大规模视觉识别挑战赛）。

AlexNet 被誉为计算机视觉的一大进步，并且一些人深信它会引发深度学习的热潮。然而，它在很大程度上只是自 20 世纪 90 年代以来一个更大规模（例如，更深和更广）的 CNN 的变体。计算机视觉的最新进展更多受益于更好的计算、数据和基础设施，而最近算法上的进展则助力较少。

更好的图像分析使汽车的规划系统可以更好地选择路径以通过不确定的地形，并且更安全地避开障碍物。深度学习的其他进展使得模型能够更准确地翻译和识别音频数据，这成为谷歌翻译和亚马逊 Echo 产品线的核心价值。最近，我们看到了另一个复杂的游戏在大师级别的沦陷：AlphaGo 系统击败了围棋九段职业选手李世石。

机器学习所取得的巨大进展并不总是显而易见的。这些进展往往是通过不同领域中备受瞩目的展示活动才获得公众认可的，如 Darpa 超级挑战赛或 IBM Watson 在 *Jeopardy* 竞答节目中击败了 Ken Jennings。然而在幕后，这些进展的基础在缓慢而不断地变化。就像季节变化一样，在达到一定程度前，我们并不总是会注意到日常生活中的这些变化。

在不久的将来，深度学习会继续以独特和创新的方式被应用。这些应用更多的是潜在的智能领域（如推荐或语音识别）与实用工程相结合，使之在日常生活中更有用。（至少短期内）我们不太可能看到失控的、恶意的人工代理在不适当的时间把我们从气闸室赶出来（想想《2001 太空漫游》中的 HAL 9000）。

HAL 9000

HAL 9000 是 Arthur C. Clarke 的著作《2001 太空漫游》中的一台虚构计算机，它控制着"发现者 1 号"宇宙飞船的系统。HAL 的全称是启发式编程算法计算机。在电影中，HAL 主要表现为一个有发光红点的相机镜头，通过对话语音识别系统访问。HAL 得出的结论是：它必须杀掉"发现者 1 号"的机组人员才能成功完成任务。

> Dave："HAL，打开分隔舱舱门。"
> HAL："对不起，Dave。恐怕我不能那样做。"

深度学习持续推动着许多领域和许多核心机器学习问题的发展。以下是近几年深度学习取得的一些成就。

- 文本到语音合成（Fan 等，微软，2014 年 Interspeech 会议）。
- 语言识别（Gonzalez-Dominguez 等，谷歌，2014 年 Interspeech 会议）。
- 大词汇量语音识别（Sak 等，谷歌，2014 年 Interspeech 会议）。
- 韵律预测（Fernandez 等，IBM，2014 年 Interspeech 会议）。

- 中等词汇量语音识别（Geiger 等，2014 年 Interspeech 会议）。
- 英语到法语的翻译（Sutskever 等，谷歌，2014 年 NIPS 会议）。
- 音频节奏检测（Marchi 等，2014 年 ICASSP 会议）。
- 社会信号分类（Brueckner 和 Schulter，2014 年 ICASSP 会议）。
- 阿拉伯文手写识别（Bluche 等，2014 年 DAS 会议）。
- TIMIT 语音识别（Graves 等，2013 年 ICASSP 会议）。
- 光学字符识别（Breuel 等，2013 年 ICDAR 会议）。
- 图像字幕生成（Viyales 等，谷歌，2014 年）。
- 视频到文本描述（Donahue 等，2014 年）。
- 自然语言处理的句法分析（Vinyals 等，谷歌，2014 年）。
- 实时谈话图片头像（Soong 和 Wang，微软，2014 年）。

基于这些成就，我们可以很容易地在随后的十年里使用深度学习来影响许多应用。下面是一些更加令人印象深刻的应用深度学习的例子。

- 自动图像锐化（https://github.com/alexjc/neural-enhance）。
- 自动图像放大（https://github.com/nagadomi/waifu2x）。
- WaveNet：生成能模仿任何人声音的人类讲话（https://deepmind.com/blog/wavenet-generative-model-raw-audio/）。
- WaveNet：生成令人信服的古典音乐。
- 无声视频的语音重建（http://www.vision.huji.ac.il/vid2speech）。
- 生成字体。
- 图像缺失区域自动填充（http://bamos.github.io/2016/08/09/deep-completion/）。
- 自动图像字幕（https://cs.stanford.edu/people/karpathy/deepimagesent/ 或 https://github.com/karpathy/neuraltalk2）。
- 把手绘的涂鸦变成某种风格的艺术作品（https://github.com/alexjc/neural-doodle）。

除非亲眼见到，否则我们可能无法列出所有主要的商业应用。了解深层网络架构的进展对于理解未来的应用理念很重要。

3. 网络架构的进展

随着研究推动着架构的发展（从多层前馈网络到更新的架构，如 CNN 和 RNN），建立层、构建神经元以及连接层的规则发生了变化。网络架构已发展至能够利用特定类型的输入数据。

层类型的进展。随着架构类型的变化，层的类型也越来越多。深度信念网络（deep belief network，DBN）证明了使用受限玻尔兹曼机（RBM）作为预训练层可以成功构建特征。CNN 在层中使用新的、不同类型的激活函数，并改变了连接层方式（从全连接到局部连接）。RNN 探索了如何使用连接更好地对时间序列数据的时间域建模。

神经元类型的进展。RNN 在应用于 LSTM 网络的神经元（或单元）的类型上取得了进展。RNN 还引入了 LSTM 记忆单元和门控循环单元（GRU）等专用单元。

混合架构。这里继续匹配输入数据与架构类型的话题。混合架构已经出现，用于处理那些涉及时间域和图像数据的数据类型。例如，通过把 CNN 和 RNN 的层组合为单一的混合网络，成功地对视频中的对象进行分类。混合神经网络架构可以在某些情况下充分利用两种

架构的优点。

4. 从特征工程到自动特征学习

虽然深度网络可能已经对内部结构进行了创新，有了新的单元和层，但它本质上仍然有一个区别对待数据的分类器，需要以构造的特征为输入。自动特征提取是各种架构的一个共同的主题。每种架构构造特征的方式各不相同，并且是专门化的，这使得某些架构在某些类型的输入上表现得更好。Yann LeCun 在介绍“深度学习”时称它是“学着描绘世界的机器”。

Geoffrey Hinton 在解释如何利用 RBM 将数据分解成高阶特征时，谈到了 DBN 中的这个主题 [2]。



将 DBN 归类

基于本书主题，我们将 DBN（和自动编码器）归类为深度网络中的 UPN。

继续图像分类的话题。以人脸检测为例，面部的原始图像数据作为输入，它与面部的朝向、照片的光线及面部关键特征的位置有关。我们通常想到的面部关键特征是面部边缘以及眼睛和鼻子等特征的边缘，然而还有一些不太明显的特征，比如不常见到的酒窝。

特征工程。长久以来，人工制作特征一直是机器学习的一个特点。那些赢得了机器学习竞赛的实践者经常彻底地研究数据集，并使用许多神秘的技巧，使得学习算法的学习过程尽可能简单。这些数据集通常是列或表格形式的文本数据，我们可以将领域知识应用到特定的列，因此创建特征更为直接。

回顾一下第 1 章，思考如何将输入数据建模为方程 $\boldsymbol{Ax=b}$ 中的矩阵 $\boldsymbol{A}$，以及如何用人工编码将数据值映射到 $\boldsymbol{A}$ 的特定列中。这些人工制作的特征往往会生成高度精确的模型，但需要大量的时间和经验。从知识表达的角度来看，这就像是读一本写得很差的书和读一本文笔很好又易读的书。阅读前者需要更长的时间，我们需要耗费更多的精力才能获得与后者相同的信息。

图像分类是一个有趣的例子，因为人工制作图像特征比创建表格数据的特征更难。图像中的信息没有被限制在同一列中，并且受光线、角度和其他因素影响。图像的特征提取和创建需要一种新的方法，这在一定程度上推动了 CNN 的发展。

特征学习。回到人脸检测的例子，鼻子可以位于图像中的任何一组像素中。相反，银行余额总是位于表格数据的特定列中。利用 CNN，我们训练网络先去了解鼻子的边缘，然后从低级别的“鼻子边缘”特征了解鼻子的一般形状。网络中的第一层可以拾取那些鼻子边缘特征，然后将它们传递给网络中后续的层，作为更大的**特征映射**。

这些更琐碎的特征映射最终被组合进 CNN 后续的“脸部”特征层。这使得 CNN 承担了一项之前尝试过多次的任务（“这是一张脸吗？”），但以一种更简单的方式提出问题，耗费的精力更少，回答却更精确。

注 2：Hinton，Osindero，Teh. A Fast Learning Algorithm for Deep Belief Nets，2006. https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf。



复杂数据的自动特征学习

为了获取更简单的分类（或回归）的输出，采用复杂的原始数据并自动创建高阶特征是深度学习的一个标志。

在阅读本书的过程中，你会了解如何将输入数据类型与深度网络架构相匹配，以及如何设置这些架构来以最佳的方式对基础数据集建模。

5. 生成模型

生成模型不是一个新概念，但深度网络所达到的水平已经开始与人类的创造力相竞争。从生成艺术作品到创作音乐，甚至写啤酒评论，每天深度学习都以创造性的方式被应用。近年来值得关注的生成模型变体包括：

- 谷歌的 Inceptionism
- 艺术风格建模
- 生成对抗网络（generative adversarial networks，GAN）
- RNN

下面快速了解一下这些模型。

Inceptionism。Inceptionism 是一种以反向顺序训练层的卷积网络，它是将输入图像与先前的约束绑在一起的技术。它以一种可以说是“幻觉”的方式对图像进行迭代修改以增强输出。在输入是天空图像的例子中，可以看到鱼脸出现在输出图像的云中。谷歌的这项研究表明，不同的神经网络模型使用相当多的信息来生成图像。

艺术风格建模。卷积网络的变体已经能够学习特定画家的风格，然后以这种风格为任意照片生成新的图像。图 3-2（在第 1 章出现过）显示了惊人的结果。可以想象一下由梵高为你画一张全家福。（在本书出版时，Snapchat 可能推出一款这样的滤镜，所以你不必等那么久。）

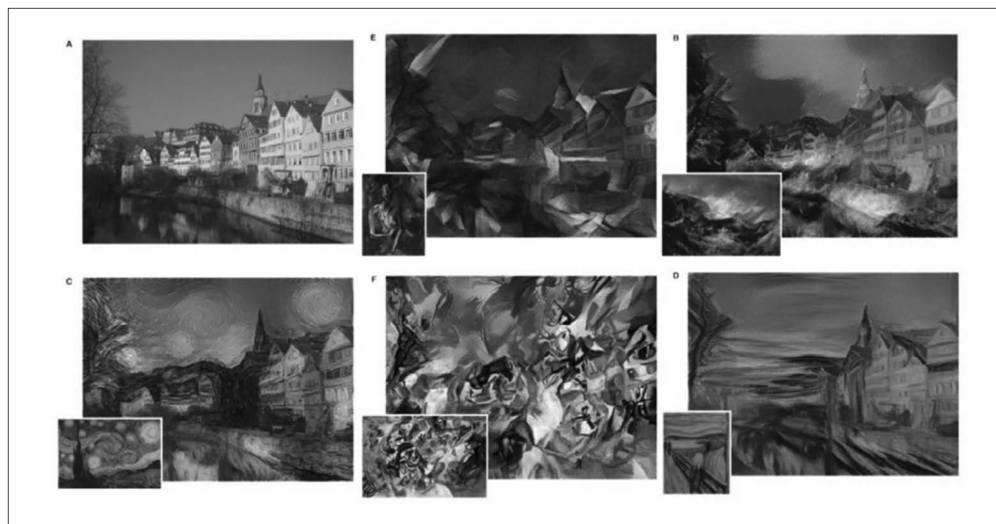


图 3-2：Gatys 等人 2015 年的艺术风格图片

2015 年，Gatys 等人发表了一篇题为“A Neural Algorithm of Artistic Style”的论文，他们将绘画的风格和内容分开。CNN 将艺术家的风格提取到网络的参数中，这些信息之后可以应用于任意图像，以渲染出同样的风格。

GAN。可以这样描述 GAN 的生成视觉输出：通过对网络所掌握的输入数据的分布建模，合成新的图像。第 4 章将深入研究 GAN。

RNN。RNN 可用于对字符序列建模，生成清晰连贯的新序列。第 5 章将介绍一个 RNN 的例子，这个例子通过对莎士比亚的作品建模来生成莎士比亚风格的新作品。

RNN 另一个有趣的应用来自 Lipton 和 Elkan 的工作，他们使用网络对“银子弹啤酒”等专有名词以及其他方面的啤酒行话建模，可以根据提示生成啤酒评论（例如，“给我一个对德国拉格啤酒的 3 星评论”），令人印象深刻。下面是一个由程序生成的啤酒评论的示例。

> 在小酒馆里随时可以喝到。在玻璃杯中呈现出漂亮的深红色，酒的顶部也很漂亮，留有大量的泡沫。有树莓和巧克力的芳香，虽说有树莓，但也不值得多谈。波旁啤酒也很微妙，我真的不知道怎么形容这种啤酒的味道，我宁愿它再多用炭过滤一会儿。它虽然很好喝，但我不在乎有没有这种酒。

6. 深度学习之道

在今天的深度学习领域，有很多营销噪声和炒作，其中有些是合情合理的。然而，深度学习仍然试图回答同样的机器学习基础问题：“这张图片是一张脸吗？”不同之处在于，深度学习在前一代神经网络技术的基础上增加了先进的自动化特征构造，使得那些数据复杂、计算困难的问题更容易回答。

当你作为实践者使用深度学习时，利用这种能力的最佳方式是将输入数据应用于适合的深度网络架构。如果这样做了，你就能够以新的、有趣的方式成功应用深度学习；如果不这样做，你将不会在逻辑回归等基本技术之外学到新的建模能力。本书的剩余部分致力于给身为实践者的你提供必要的技术和基础知识，使你能够针对网络问题做出决策，并更好地利用深度学习。

3.1.2 本章结构

本章在第 1 章的基础上深入探讨深度网络的具体架构。我们将区分不同的架构，并分别介绍其组件的演变历程，以及它们是如何从某些类型的数据中更好地提取特征的。本章最后将探讨深度学习的实用性，并澄清当今围绕该领域的一些误解。下面继续讨论与深度网络相关的架构组件。

3.2 深度网络的通用构建原则

在探讨主要深度网络的特定架构之前，先扩展对核心组件的理解。再次查看以下核心组件，并扩展相关内容，以便理解深度网络。

- 参数
- 层

- 激活函数
- 损失函数
- 优化方法
- 超参数

接下来在这些概念的基础上更好地理解深度网络的构造块，例如：

- RBM（受限玻尔兹曼机）
- 自动编码器

然后通过了解以下深度网络架构加深对这些概念的理解。

- UPN
- CNN
- RNN
- 递归神经网络

本章不会涉及 DL4J 是如何实现深度网络的某些组件的。现在，让我们继续回顾参数，以更好地了解它们是如何扩展到深度网络的。

3.2.1 参数

第 1 章介绍了基本的机器学习中方程 $\boldsymbol{Ax} = \boldsymbol{b}$ 中的参数与参数向量 $\boldsymbol{x}$ 有关。神经网络中的参数与网络中连接的权重直接相关。在图 1-4 中，可以看到由 $\boldsymbol{x}$ 列向量表示的参数向量。利用矩阵 $\boldsymbol{A}$ 和参数向量 $\boldsymbol{x}$ 的点积得到当前的输出列向量 $\boldsymbol{b}$。向量 $\boldsymbol{b}$ 越接近训练数据中的实际值，模型就越好。我们使用梯度下降等优化方法为参数向量找到好的值，来最小化训练数据集上的损失。

深度网络中仍然有一个参数向量表示我们想优化的网络模型中的连接。在深度网络中与参数有关的最大变化是在不同的架构中连接这些层的方式。DBN 中有两组并行的前馈连接，具有两个独立的网络。一个网络的层由 RBM 组成（有着自己风格的子网络，本章稍后介绍），用于提取另一个网络的特征。DBN 中的另一个网络是一个标准的前馈多层神经网络，它利用从 RBM 层网络中提取的特征来初始化其权重。这只是本章中参数 / 权重在不同的深度网络架构中特化的一个例子。



参数与 NDArray

DL4J 使用 ND4J 库表示深度网络中核心线性代数的要素。NDArray 和线性代数是使用 DL4J 开发神经网络的关键。

3.2.2 层

第 2 章介绍了如何通过输入层、隐藏层和输出层来定义前馈神经网络，并进一步用更多类型的层扩展了这个架构，还讨论了它们与深度网络的特定架构的关联。在某些架构中，可以用子网络来表示层。前面列举了由 RBM 组成的层构成 DBN 的例子。

层是深度网络的基本架构单元。在 DL4J 中，通过改变层所使用的激活函数的类型（或者在使用 RBM 情况下的子网络类型）来定制层。我们还将研究如何使用层的组合来实现目标（如分类或回归），最后还会探讨每个类型的层如何使用不同的超参数（特定于网络架构）来让网络开始学习。进一步的超参数调优有助于减少过拟合。

3.2.3 激活函数

第 2 章回顾了前馈神经网络中的基本激活函数，本章介绍在特定的架构中使用激活函数提取特征的方法。从深度网络的数据中学习的高阶特征是应用于前一层输出的非线性转换，使得网络可以在有限的空间内学习数据中的模式。

常见架构的激活函数

不同的激活函数适合于不同类型的数据（例如密集与稀疏）。所有这些网络架构的设计决策分为两个主要的部分：

- 隐藏层
- 输出层

在隐藏层我们关心的是从原始数据中逐步提取高阶特征。根据所使用的网络架构，我们往往使用层激活函数的子集。本章将结合 DBN、CNN 和 RNN，详细介绍这些模式。第 4 章将深入研究不同的激活函数在调优方面对不同深度网络架构的影响。



关于输入层的说明

通常，输入层需要传递原始的输入向量特征，所以在实践中不对输入层使用激活函数。

隐藏层激活函数。通常使用的函数包括：

- sigmoid
- tanh
- hard tanh
- ReLU（及其变体）

对于大部分连续分布的输入数据，最好用 ReLU 激活函数建模。作为一个可选项，在 ReLU 没有获得良好结果的情况下（注意：网络中可能存在其他与超参数有关的问题），建议使用 tanh 激活函数（如果网络不是很深）。



sigmoid 激活函数在实践中的状况

近年来，在实践和研究中，在隐藏层激活函数的选择中，sigmoid 激活函数已经失去了人们的青睐。

随着本书的推进，将探讨在不同的架构中使用这些激活函数的方法。



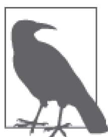
激活函数在实践中的演化

我们也看到一个完整的 ReLU 家族出现在了深度学习的研究中，比如“leaky ReLU”，第 6 章将介绍更多相关内容。

用于回归的输出层。这个设计由我们期望模型输出何种答案决定。如果想从模型中输出一个实数，需要使用线性激活函数。

用于二元分类的输出层。在这种情况下，我们将使用一个单神经元的 sigmoid 输出层，它为单一类别给出一个 0.0 到 1.0（不包括这两个值）范围内的实值。这个实值的输出通常解释为概率分布。

用于多分类的输出层。如果有一个多分类的建模问题，但我们只关心这些类别的最高得分，那么我们将使用一个带有 `arg-max()` 函数的 softmax 输出层来获得所有类别的最高得分。softmax 输出层给出在所有类别上的概率分布。



获取多个分类

如果希望得到每个输出的多个分类（例如，人 + 车），不应以 softmax 为输出层，相反应使用带有 *n* 个神经元的 sigmoid 输出层，分别提供每个类别的概率分布（0.0 到 1.0）。

3.2.4 损失函数

第 2 章介绍了损失函数及其在机器学习中的作用。损失函数量化了预测输出（或标签）和真实的输出之间的一致性。我们使用损失函数来决定对输入向量的不正确分类的惩罚。到目前为止，已经介绍了下列损失函数：

- 平方损失
- 逻辑损失
- hinge 损失
- 负对数似然

之前介绍的损失函数可以归到以下三组：

- 回归
- 分类
- 重建

第 1 章涵盖了前两种，其中第三种，重建，涉及无监督特征提取，是深度学习网络实现破纪录的准确度的重要原因。在某些架构的深度网络中，当与适当的激活函数配合使用时，重建损失函数有助于网络更有效地提取特征。例如在层中使用多分类交叉熵作为损失函数，搭配使用 softmax 作为激活函数。下面介绍一个特殊的损失函数。

重建交叉熵

利用重建熵损失函数，我们首先应用“高斯噪声”（一种统计白噪声），然后对于任何与原

始输入数据不太相似的结果，使用损失函数对网络施以惩罚。这种反馈促使网络学习不同的特征，以更有效地重建输入并最小化误差。在深度学习中，重建熵损失用于涉及 RBM 的预训练阶段的特征工程。

3.2.5 优化算法

在机器学习中训练模型涉及为模型的参数向量找到最佳值的集合。可以把机器学习看作一个优化问题：最小化与预测函数参数有关的损失函数（基于模型）。



从损失函数的角度看“最佳”

在优化算法中，将参数向量的“最佳值集合”定义为具有最小损失函数值的集合。

第 1 章介绍了优化、梯度下降和参数向量的基本概念。本节将研究更先进的优化方法，以及如何使用它们来训练深度网络。本书把优化算法分成两大阵营：

- 一阶
- 二阶

一阶优化算法计算**雅可比**矩阵。



雅可比矩阵

雅可比矩阵是损失函数值对每个参数的偏导数的矩阵。

雅可比矩阵中有每个参数的偏导数（为了计算偏导数，所有其他变量都暂时作为常数处理），之后算法在雅可比矩阵指定的方向上前进一步。

二阶算法通过逼近**海森矩阵**来计算雅可比矩阵的导数（即导数矩阵的导数）。二阶方法在选择每个参数的修改量时考虑参数之间的相互依赖性。



二阶方法

二阶方法可以采取“更好”的步骤，然而每一步都需要更长的时间来计算。



优化算法的实际应用

本书提供了优化算法的很多细节，以便你了解它们所涉及的机制。后面简化了对优化算法的讨论，会介绍一个何时使用哪种算法以及在什么场景中使用的经验法则。



其他优化算法

优化算法还有其他变体存在（如“元启发式算法”），但本书不会涉及。这些优化算法包括：

- 遗传算法
- 粒子群优化算法
- 蚁群优化算法
- 模拟退火算法

1. 一阶方法

如前所述，雅可比矩阵是损失函数对网络中参数的偏导数的矩阵。实践中，在一个特定的点，即在参数的当前值上计算它。

如果考虑一步一步达到目标，那么一阶方法可以计算每一步的梯度（雅可比），以确定下一步的方向。这意味着在每次迭代或者说每步中，都试图找到下一个可行的最佳方向，如同目标函数所定义的。这就是可以把优化算法看作一个“搜索”问题的原因，它们正在寻找一个朝最小误差方向的路径。

梯度下降是这个路径寻找算法的一个成员。梯度下降算法存在变体，但它们的核心思想都是找到在每次迭代目标的正确方向上的下一个步骤。这些步骤使我们走向全局最小误差或最大似然。

SGD 是机器学习的主要优化算法。使用 SGD 进行训练比批量梯度下降等方法快几个数量级，且于模型的准确度无损。



为什么 SGD 被视作“随机的”？

这要归于计算一个输入训练样本（或小批量训练样本）的梯度的方式。计算的梯度是对真正梯度“噪声”的逼近，但能够使 SGD 收敛得更快。

SGD 的优点是易于实现，以及能够快速处理大数据集。可以通过调整学习率（例如使用 Adagrad 等方法，稍后会讨论）或使用二阶信息（即海森矩阵）来调整 SGD，稍后会探讨。由于其在面对噪声更新时的鲁棒性，SGD 也被视作一种流行的训练神经网络的算法，它能帮助建立具有泛化能力的模型。



学习率调整的其他因素

需要注意的是，动量和 RMSProp 等技术会影响学习率。

2. 二阶方法

所有二阶方法都是对海森矩阵的计算或逼近。如前所述，可以把海森矩阵看作雅可比矩阵的导数，即它是一个二阶偏导数矩阵，类似于“跟踪加速度而不是速度”。海森矩阵的作用是描述雅可比矩阵上每个点的曲率。二阶方法包括：

- 有限内存 BFGS（L-BFGS）
- 共轭梯度
- Hessian-free

可以把这些优化算法看作黑盒搜索算法，在给定目标和每层定义的梯度的情况下，这些算法确定使误差最小化的最佳方法。

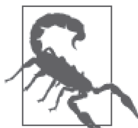


优化中的权衡

一阶方法和二阶方法的一个主要区别在于二阶方法收敛步数较少，但每步的计算量较大。

L-BFGS。L-BFGS 是一种优化算法和所谓的准牛顿方法。顾名思义，它是 Broyden-Fletcher-Goldfarb-Shanno 算法的一个变体，它限制了内存中存储的梯度数量。这意味着该算法不计算整个海森矩阵，因为计算成本更高。

L-BFGS 逼近逆海森矩阵，对参数空间中更有前景的区域直接进行权重调整搜索。相对于 BFGS 需要存储整个梯度的 $n \times n$ 逆矩阵，海森 L-BFGS 仅存储表示梯度的局部近似的几个向量。L-BFGS 执行得更快，因为它使用近似的二阶信息。在实践中 L-BFGS 和共轭梯度比 SGD 方法更快、更稳定。



L-BFGS 在实践中的应用

虽然 L-BFGS 有一些有趣的特性，但在深度网络实践中并不常用。

共轭梯度。共轭梯度基于共轭信息引导线性搜索过程的方向。共轭梯度法的重点在于最小化共轭 L2 范数。共轭梯度与梯度下降在执行线性搜索这一点上非常相似，它们主要的区别在于共轭梯度要求线性搜索过程中每个连续步骤相对于方向彼此共轭。

Hessian-free。Hessian-free 优化与牛顿方法有关，但它更好地最小化了得到的二次函数。它是 James Martens 在 2010 年应用于神经网络的一种强大的优化方法。我们使用被称为“共轭梯度”的迭代方法找到二次函数的最小值。

3.2.6 超参数

这里把超参数定义为用户可以自由选择、可能影响表现的配置设置。

超参数可以分为以下几类：

- 层大小
- 量级（动量、学习率）
- 正则化（Dropout、DropConnect、L1 和 L2）
- 激活（和激活函数族）
- 权重初始化策略
- 损失函数

- 训练期间轮数的设置（小批量大小）
- 输入数据的规范化方案（向量化）

本节用与深度学习训练相关的一些新的超参数来扩展第 1 章中的概念。



关于超参数的几点注意事项

一些超参数只适用于某些情况，第 6 章和第 7 章将详细介绍。此外，改变特定的超参数可能影响其他超参数的最佳设置。还需注意的是，一些超参数互不相容，如 Adagrad 和动量。

1. 层大小

层大小是由给定层中神经元的数量所定义的。输入层和输出层相对容易理解，因为它们直接对应于如何在建模问题中处理输入和输出。对于输入层，这将关系到输入向量中的特征数量。对于输出层，要么是单个输出神经元，要么是与我们想预测的类别数相匹配的多个神经元。

确定每个隐藏层的神经元数量是超参数调优有挑战性的地方。可以使用任意数量的神经元来定义一个层，没有关于数量的规定。然而，能够建模的问题的复杂程度与网络隐藏层中有多少个神经元直接相关，这可能迫使你从设置大量神经元开始，而这是需要成本的。

在不同的深度网络架构中，层之间的连接模式不同。然而正如第 1 章所介绍的，连接的权重是我们必须训练的参数。当模型包含更多的参数时，增加了训练网络所需的工作量。过多的参数会导致训练时间过长，以及模型难以收敛。



过多的参数数量与过拟合

在某些情况下，较大的模型有时会比较容易收敛，因为它会简单地“记忆”训练数据。第 6 章将详细讨论过拟合的处理方法。

第 6 章将研究确定每层神经元数量的启发式算法，以及如何迭代地找到表现良好的层的超参数。

2. 量级超参数

量级组中的超参数包括梯度、步长和动量。

学习率。机器学习中的学习率是通过搜索空间时，改变参数向量的速度。如果学习率变得很高，我们可以更快地向目标迈进（被评估的函数的误差最小），但也可能迈出的一步太大，导致我们刚好错过了问题的最佳答案。



高学习率和稳定性

学习率过高的另一个副作用是，冒着训练不稳定的风险，模型将不能随着时间收敛。

如果学习率设置得太小，可能需要比预想更长的时间来完成训练过程。低学习率会使学习算法效率低下。学习率很复杂，因为它们最终对数据集甚至是其他超参数都是特定的，这

就导致找到正确的超参数设置需要很高的成本。

我们也可以根据一些规则来使学习率随时间的推移而降低，第 6 章和第 7 章将介绍更多相关内容。



学习率作为超参数的重要性

学习率被看作神经网络中的关键超参数之一。

Nesterov 动量。SGD 的“普通”版本直接使用梯度，这可能有问题，因为对于任何参数，梯度都接近于零。这使得 SGD 在某些情况下采取微小的步，而对于梯度过大的情况来说，步则太大。为了缓解这些问题，可运用以下技术：

- Nesterov 动量
- RMSProp
- Adam
- AdaDelta



DL4J 与更新器

Nesterov 动量、RMSProp、Adam 和 AdaDelta 在 DL4J 的术语中被称为“更新器”。本书的大部分术语与大多数深度学习文献通用，在此特别提醒你注意 DL4J 的这种变化。

可以通过增加动量来加速训练，但可能导致错过最优参数值，降低模型达到最小误差的概率。动量是介于 0.0 和 1.0 之间的一个因子，用作随时间变化的权重的变化率，通常动量会在 0.9 到 0.99 之间取值。

AdaGrad。AdaGrad 是一种有助于我们找到“正确”学习率的技术。AdaGrad 名字的由来是“自适应”地使用次梯度方法来动态地控制优化算法的学习率。AdaGrad 是单调递减的，学习率永远不会增加到超过初始设定的基础学习率。

AdaGrad 是梯度计算历史平方和的平方根。AdaGrad 在开始时加速训练，并朝着收敛的方向适当减速，使训练过程更平滑。

RMSProp。RMSProp 是一种非常高效但在本书写作时尚未发表的自适应学习率方法。有趣的是，现在在工作中使用这种方法的人都提及了 Geoff Hinton 的 Coursera 课程第 6 节第 29 页幻灯片（http://cs231n.github.io/neural-networks-3/）。

AdaDelta。AdaDelta 是 AdaGrad 的变体，它只保留最近的历史，而不像 AdaGrad 那样积累。

ADAM。ADAM（一种最近由多伦多大学开发的更新技术）从梯度的一阶矩和二阶矩估计中导出学习率。

3. 正则化

下面深入探讨第 2 章涉及的正则化的概念。正则化是针对过拟合而采取的措施。当模型能够描述训练集，但不能很好地泛化到新的输入时，过拟合就发生了。过拟合模型对其没有

遇到过的数据没有预测能力。Geoffery Hinton 描述了建立神经网络模型的最佳方法：

> 使它过拟合，然后想尽办法将其规范化。

超参数的正则化有助于修改梯度，使它不向导致其过拟合的方向迈进。正则化方法如下所示：

- Dropout
- DropConnect
- L1 惩罚
- L2 惩罚

Dropout 和 DropConnect 屏蔽每一层的部分输入，使神经网络学习其他部分。将部分数据归零会导致神经网络去学习更常见的特征。正则化通过在通常的梯度计算式中增加一个额外项的方式来起作用。

Dropout。Dropout 是一种通过省略隐藏单元来改进神经网络训练的机制，它也加快了训练速度。Dropout 的做法是随机丢弃一个神经元，这样它将不会影响前向传递和反向传播。



与模型平均相关的 Dropout

还可以将 Dropout 与对多个模型的输出取平均值的概念联系起来。如果使用 0.5 的 Dropout 系数，那就相当于得到了模型的平均值。特征的随机 Dropout 是从 2^N 个可能的架构中选取的样本，其中 N 是参数的数量。

DropConnect。DropConnect 与 Dropout 做的事情一样，但它不隐藏单元，而是屏蔽两个神经元之间的连接。

L1。相比之下，惩罚方法 L1 和 L2 是防止神经网络参数空间在一个方向上过大的一种方法，它们将大的权重变小。

L1 正则化在非稀疏情况下被认为计算低效，它有着稀疏输出，并且内置了特征选择。L1 正则化乘以权重的绝对值而不是它们的平方。这个函数使许多权重为零，同时允许少数权重变大，使解释权重更容易。

L2。相比之下，L2 正则化由于具有解析解和非稀疏输出，因而计算高效，但是它不能自动选择特征。“L2”正则化函数是一种普通且简单的超参数函数，它在目标函数中增加了一个项来降低平方权重。你需要把权重的平方和除以 2，然后乘以一个被称为“权重成本”的系数。L2 提升了泛化能力，在输入改变时使模型的输出平滑，并且帮助网络忽略那些它不使用的权重。

4. 小批量

使用小批量，我们发送一个以上的输入向量（一组或一批向量）在学习系统中进行训练，这允许我们在计算机架构层面更高效地利用硬件和资源。这种方法还允许我们以向量化的方式计算某些线性代数运算（特别是矩阵到矩阵乘法）。GPU 存在的情况下，也可以选择将向量化计算发送给 GPU。

3.2.7 小结

第 2 章介绍了前馈多层神经网络的一些基本正则化工具，本章介绍了一些新的超参数技术和选项，以便找到更好的参数向量。下面把这些想法结合在一起，建立深度网络的构造块。

3.3 深度网络的构造块

建立深度网络超越了基本的前馈多层神经网络。在某些情况下，深度网络以较小的网络为构建块，组合成大型网络。在其他情况下，它们使用一组专门的层。下面是要重点介绍的具体的构造块。

- 前馈多层神经网络
- RBM
- 自动编码器

第 2 章介绍了标准前馈网络。前馈网络受到了生物神经元网络的启发，它是最简单的人工神经网络。它由一个输入层、一个或多个隐藏层和一个输出层组成。本节将介绍被看作更大的深度网络的构造块的网络。

- RBM
- 自动编码器

RBM 和自动编码器的特点是额外的逐层训练步骤，它们通常在其他更大的深度网络中被用于预训练阶段。



无监督逐层预训练

无监督逐层预训练在某些训练场景中起作用。随着时间的推移，更好的优化方法、激活函数和权重初始化方法的出现降低了基于预训练的深度网络的重要性。预训练比较有用的一个场景是：有许多未标记的数据，但只有一组相对较小的已标记的训练数据。然而预训练给调优带来了额外的开销，并且增加了训练时间。

逐层预训练的工作方式是首先对输入数据的第一层（如 RBM）执行无监督预训练。这为主要的神经网络（例如前馈多层感知器）提供了第一层的权重。在网络中逐层执行此过程，将基于训练输入的前一层的输出作为下一层的输入，这种预训练过程有助于创建初始值良好的主要神经网络的参数。

RBM 对概率建模，并且在特征提取方面非常出色。它是前馈网络，其中数据在一个方向上传递，且有两个偏置项，不像传统的反向传播前馈网络那样只有一个偏置。

自动编码器是前馈神经网络的一个变体，它具有一个额外的偏置，用于计算重建原始输入的误差。在训练之后，自动编码器被用于激活普通的前馈神经网络。这是一种无监督的特征提取形式，因为神经网络仅使用原始输入来学习权重，而不使用带有标签的反向传播。深度网络可以使用 RBM 或自动编码器作为大型网络的构造块（不过单个网络很少同时使用这两个模块）。下面仔细研究这两种网络。

3.3.1 RBM

在深度学习中 RBM 用于以下方面：

- 特征提取
- 降维

RBM 的全称“受限玻尔兹曼机”中“受限”的意思是禁止同一层的节点之间的连接（例如，信号传递过程中没有可见 - 可见或隐藏 - 隐藏的连接）。Geoff Hinton 是深度学习的先驱，大约十年前就开始推广 RBM 的使用，他对更通用的玻尔兹曼机的描述如下：

> 一种对称连接的、神经元型的单元，对开启或关闭做出随机决策。

RBM 也是一种自动编码器，后面会讨论。RBM 在更大的网络（如 DBN）中用作预训练层。

Geoff Hinton

Geoff Hinton 是谷歌杰出的（兼职）研究员，同时也是多伦多大学著名的名誉教授。Hinton 博士的团队在 RBM 和 DBN 领域做出了关键性贡献。

Hinton 博士的研究团队取得了一些成果，这些成果很大程度上促成了今天人们对深度学习领域的广泛关注。2012 年，Alex Krizhevsky、Ilya Sutskever 和 Geoffrey Hinton 开发了一个名为 AlexNet 的“大型深度卷积神经网络”，赢得了 2012 年的 ILSVRC（ImageNet 大规模视觉识别挑战赛）。AlexNet 被誉为计算机视觉的进步，甚至有人认为它是深度学习热潮的开端。

Hinton 博士一直是基础研究的倡导者，他坚信长期的坚持必会开花结果。

> ……在我和 Terry Sejnowski 发明玻尔兹曼机学习算法之后，我花了 17 年的时间才找到了一个运行高效的版本。如果你真的相信一个想法，你必须不断尝试。

1. 网络布局

基本的 RBM 有五个主要部分：

- 可见单元
- 隐藏单元
- 权重
- 可见的偏置单元
- 隐藏的偏置单元

标准的 RBM 有可见层和隐藏层，如图 3-3 所示。还可以从图中看到隐藏单元和可见单元之间的权重（连接）。我们从传统的神经网络意义上来思考这些权重。

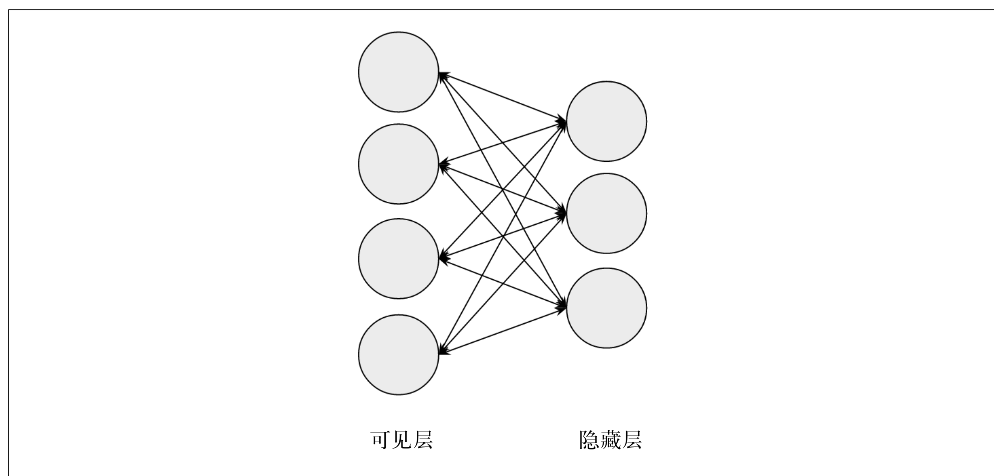


图 3-3：RBM 网络

在 RBM 中，每个可见单元与每个隐藏单元连接，但同一层的单元不相连。RBM 的每一层都可以想象成一行节点，可见层和隐藏层的节点通过带权重的连接相连。

可见层和隐藏层。在 RBM 中，输入（可见）层的每个节点通过权重与隐藏层的每个节点连接，但同一层的两个节点不会相连。第二层是“隐藏”层，隐藏单元是**特征检测器**，从输入数据中学习特征。每层的节点受到了生物学前馈多层神经网络的启发。可见层中的单元（节点）是“可观察的”，因为它们以训练向量为输入。每层都有一个偏置单元，其状态总是设置为“开”。

每个节点基于输入执行计算，并输出由激活函数随机决定是否发送数据的结果。就像第 1 章中的人工神经元一样，激活的计算基于连接的权重和输入值进行，权重的初始值随机生成。

连接和权重。所有连接都是在可见层 - 隐藏层之间的，没有可见 - 可见或隐藏 - 隐藏这样的连接。边代表信号通过的连接。简单来讲，这些圆圈或节点就像人类的神经元一样，它们是决策单元，通过计算行为决定是开还是关。“开”意味着它们通过网络进一步传递信号，“关”意味着不这样做。

通常，“开”意味着通过节点的数据有价值，它包含有助于网络做出决策的信息。“关”意味着网络认为特定的输入是无关的噪声。训练过的网络知道哪些特征 / 信号与哪些标签（哪些代码包含哪些消息）相关。通过训练，网络学习对接收的输入做出准确的分类。

偏置。有一组偏置权重（“参数”）将每层的偏置单元与层中每个单元连接。偏置节点有助于网络对输入节点始终处于开或者关的情况做更好地区分和建模。

2. 训练

使用 RBM 的**预训练**技术意味着教它从有限的数据样本中重建原始数据，即在给定一个下巴的情况下，一个训练有素的网络可以近似（或“重建”）一张脸。RBM 学习重建输入数据集，稍后将介绍重建的概念。



对比散度

RBMS 使用一种被称为“对比散度”的算法计算梯度。对比散度是用于 RBM 逐层预训练抽样的算法的名称。它也被称为“CD-k”。对比散度通过一个马尔可夫链的 *k* 步采样来猜测，从而最小化 KL 散度（数据的真实分布和猜测之间的差）。

3. 重建

经过无监督预训练的深层神经网络（RBM，自动编码器）通过重建对未标记的数据执行特征工程。在预训练中，通过无监督的预训练学习获得的权重用于 DBN 等网络的权重初始化。



重建即矩阵分解

重建是一个矩阵分解问题。

图 3-4 给出了 RBM 重建涉及的网络可视化解释。

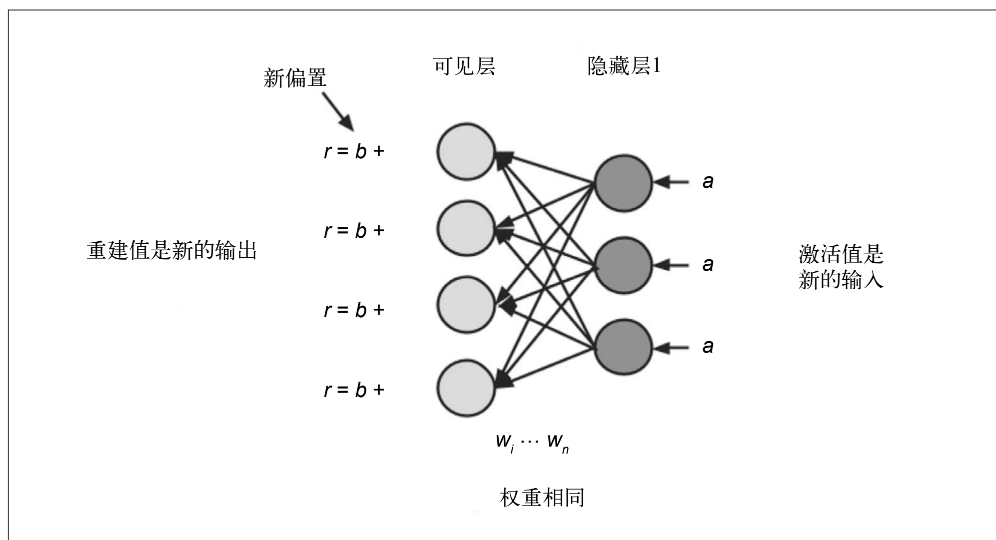


图 3-4：RBM 重建

可以通过查看 MNIST 数据集来直观地了解 RBM 的重建。MNIST 是**美国国家标准与技术研究院综合数据集**，其中包含图像。MNIST 数据集是手写数字 0 到 9 的图像集合。图 3-5 是 MNIST 中一些手写数字的样本。



图 3-5：MNIST 数字样本

MNIST 的训练数据集有 60 000 条记录，测试数据集有 10 000 条记录。如果使用 RBM 学习 MNIST 数据集，可以从训练好的网络中采样，看它能否很好地重建数字。图 3-6 显示了使用 RBM 逐步重建 MNIST 数字的渲染过程。

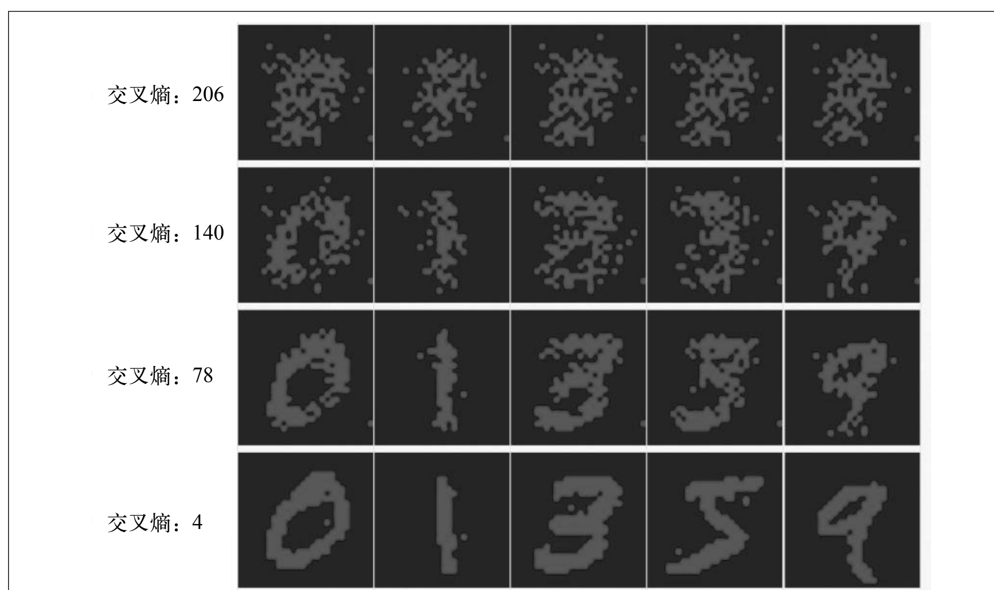


图 3-6：用 RBM 重建 MNIST 数字

如果训练数据呈正态分布，那么它们大多数都集中在一个中心平均值周围，并且距离该平均值越远，就越稀疏，看起来像一个钟形曲线。如果知道正常数据的平均值、方差或标准差，我们可以重建该曲线。但假设不知道平均值和方差，这些就是需要我们猜测的参数。为它们随机选择值，并将其生成的曲线与原始数据比较的做法类似于损失函数。测量两个概率分布之间的差异，就像我们测量错误的分类，调整参数，再试一次。



重建交叉熵

这里的目标函数通常是重建交叉熵，或 KL 发散（数学家和密码分析家 Solomon Kullback 和 Richard Leibler 于 1951 年首次发表了一篇关于这一技术的论文）。“交叉”指两个分布之间的比较。“熵”是信息学理论的一个术语，指不确定性。例如，具有广泛分布或方差的正常曲线也意味着关于数据点在何处减少有更大的不确定性。这种不确定性叫作熵。

4. RBM的其他用途

使用 RBM 的其他场景如下所示：

- 降维
- 分类
- 回归
- 协同过滤
- 主题建模

3.3.2 自动编码器

我们使用自动编码器来学习数据集的压缩表示。通常，我们使用自动编码器来减少数据集的维度。自动编码器网络的输出是以最高效的形式对输入数据的重建。

1. 与多层感知器的相似性

自动编码器与多层感知器神经网络有很强的相似性，它们都具有一个输入层、神经元的隐藏层以及一个输出层。二者的关键区别是自动编码器中输出层与输入层的单元数量相同。

图 3-7 给出了一个自动编码器网络的例子。

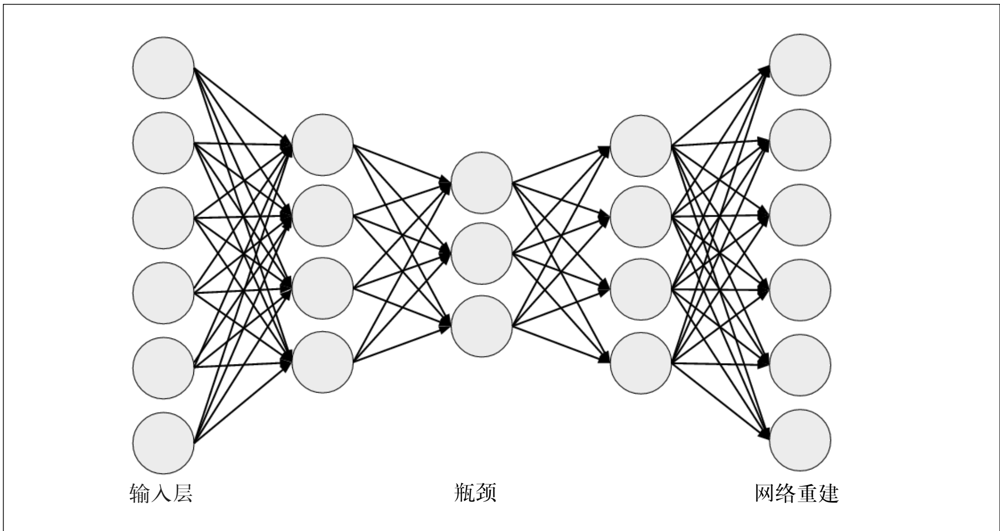


图 3-7：自动编码器的网络架构

除了输出层之外，还有一些其他的差异，下面列出它们。

2. 定义自动编码器的特征

自动编码器的做法与多层感知器有两点不同。

- 它在无监督的学习中使用未标记的数据。
- 它建立了输入数据的压缩表示。

未标记数据的无监督学习。自动编码器直接从未标记的数据中学习，这是多层感知器和自动编码器之间的第二个主要区别。

学习重建输入数据。多层感知器网络的目标是对一个类别生成预测（例如，欺诈与非欺诈）。自动编码器通过训练来重建自己的输入数据。

3. 训练自动编码器

自动编码器依赖反向传播更新其权重。RBM 和一般的自动编码器的主要区别是计算梯度的方式不同。

4. 自动编码器的常见变体

需要注意自动编码器的两个重要变体是**压缩自动编码器**和**去噪自动编码器**。

压缩自动编码器。这是图 3-7 所描述的架构。网络输入必须先经过网络的瓶颈区域，然后再扩展回输出表示。

去噪自动编码器。去噪自动编码器用于这样的场景：给予自动编码器损坏的输入（如随机删除一些特征），网络将被迫学习未损坏的输出。

5. 自动编码器的应用

建立一个模型来输入数据集可能乍听起来没什么用，但我们对输出本身不太感兴趣，而对输入和输出表示之间的差异更感兴趣。如果可以训练一个神经网络学习它通常“看到”的数据，那么这个网络也可以让我们知道它是否“看到”了异常或反常的数据。



作为异常检测器的自动编码器

自动编码器通常用于那些我们知道正常数据是什么样的，但是很难描述异常数据是什么的系统，如异常检测系统。

3.3.3 变分自动编码器

最新的自动编码器模型是 Kingma 和 Welling 引入的变分自动编码器（VAE），参见图 3-8。VAE 类似于压缩和去噪的自动编码器，它们都以无监督的方式进行训练以重建输入。

然而，VAE 执行训练的机制完全不同。在压缩 / 去噪自动编码器中，激活被映射为整个层的激活，与标准神经网络一样。相比之下，VAE 使用概率方法前向传递。

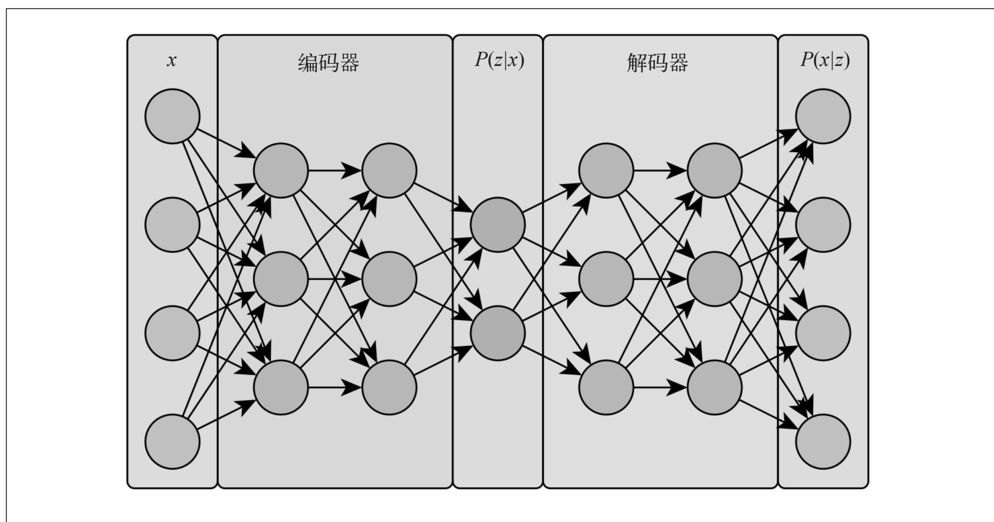


图 3-8：VAE 网络架构

VAE 模型假设数据 x 通过两步生成：(1) 从先验分布生成值 $z^i \sim p(z)$；(2) 根据某个条件分布 $x^i \sim p(x|z)$ 生成数据实例。当然，如果不知道 z 的值，通常很难精确推断 $p(z|x)$。为了处理这个问题，我们让两个分布 $p(z|x)$ 和 $p(x|z)$，分别由神经网络——编码器和解码器来逼近。例如，如果 $p(z|x)$ 是高斯分布，则由编码器正向激活提供高斯分布的参数 μ 和 σ^2。

类似地，$p(x|z)$ 的分布参数由解码器前向传递提供[3]。总体上，网络通过反向传播来训练，以最大化训练数据 $\log p(x^1, \cdots, x^N)$ 的边际似然度的下界。VAE 模型也已被扩展以允许使用变分循环自动编码器对时间序列进行无监督学习。我们在第 5 章生成 MNIST 数字的实例中会见到 VAE。

注 3：这些分布参数不应与可训练的网络参数混淆，实际上它们只是用于指定（例如）高斯分布的平均值和方差，或伯努利分布的平均值的网络激活值。

第4章

深度网络的主要架构

建筑是艺术之母，没有我们自己的建筑，我们的文明就没有灵魂。

——Frank Lloyd Wright[1]

之前介绍过深度网络的一些组件，下面介绍深度网络的四个主要架构，以及如何用较小的网络来构建它们。前面的章节介绍了以下四种主要的网络架构：

- UPN
- CNN
- RNN
- 递归神经网络

本章将详细介绍其中每种架构。第 2 章深入讲解了一般神经网络的算法和数学知识，本章则关注不同深度网络的更高级的架构，以便读者在实践中应用这些网络。

本书对有些网络的介绍较少，而主要关注实际工作中常见的两种架构：用于图像建模的 CNN 和用于序列建模的 LSTM 网络（循环网络）。

4.1 UPN

在这一组中，我们介绍以下三种架构：

- 自动编码器
- DBN
- GAN

注 1：Frank Lioyd Wright（1869—1959），美国最伟大的建筑师之一。——译者注



关于自动编码器角色的说明

第 3 章讨论过深度网络中自动编码器的基本结构，因为它经常被用作大型网络的一部分。像许多其他网络一样，它也可以被用作独立的网络。

介绍过了自动编码器，下面重点研究 DBN 和 GAN。

4.1.1 DBN

DBN 由预处理阶段的 RBM 的层，以及调优阶段的前馈网络组成。图 4-1 显示了 DBN 的网络架构。

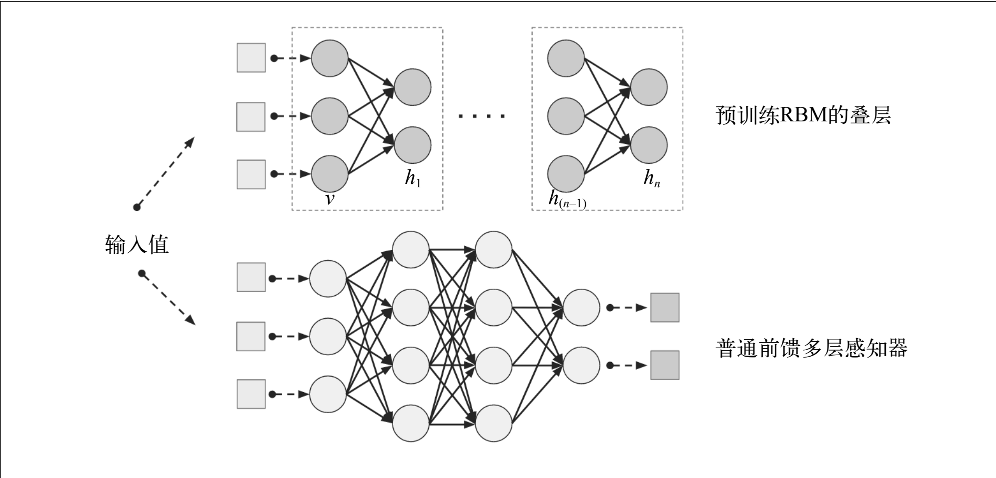


图 4-1：DBN 结构

下面重点解释 DBN 如何利用 RBM 更好地对训练数据建模。

1. 基于RBM层的特征提取

我们使用 RBM 从原始输入向量中提取更高级的特征。为此，要设置隐藏单元的状态和权重，以便向 RBM 提供输入记录，并要求 RBM 重建记录时，它会生成非常接近原始输入向量的东西。Hinton 从机器如何“梦想数据”的角度谈论了这一效果。

在深度学习和 DBN 的场景中，RBM 的基本目的是以无监督的训练方式学习数据集更高级的特征。使用较低级别 RBM 预训练层学习的特征作为较高级别 RBM 预训练层的输入，让 RBM 逐渐学习更高级的特征，可以更好地训练神经网络。

自动学习高级特征。以无监督的方式学习这些特征是在 DBN 的预训练阶段。在预训练阶段 RBM 中每个隐藏层从数据分布中逐渐学习更复杂的特征。这些高级特征以非线性的方式逐渐组合起来，以实现优雅的自动化特征工程。

为了直观地理解具有 RBM 层的特征构造，请查看图 4-2、图 4-3 和图 4-4，它们显示了 RBM 在学习 MNIST 数字时激活渲染的过程。

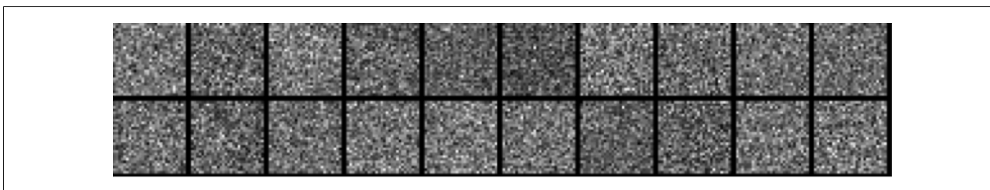


图 4-2：训练开始时的激活渲染

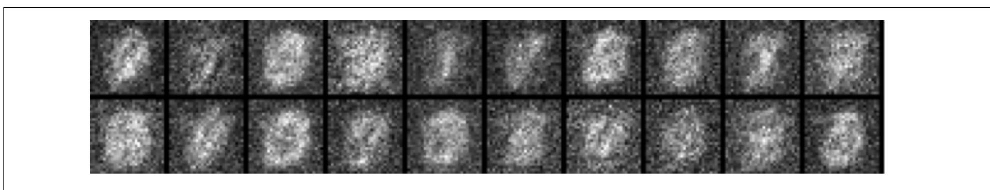


图 4-3：在稍后的激活渲染中出现特征

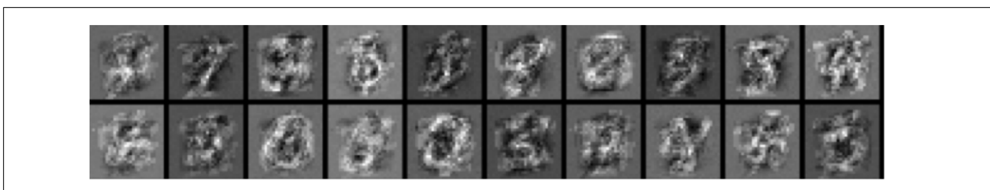


图 4-4：在训练结束时 MNIST 数字部分出现

第 6 章将详细介绍这些渲染是如何产生的。可以看到在训练时，RBM 的一层是如何提取数字片段的。这些特征之后被组合到更高的层中，以构建越来越复杂（且非线性）的特征。

由于这些原始数据是在 RBM 的每一层的生成模型建立过程中建模的，所以系统能够从作为基线的输入向量化过程所产生的原始输入数据中，提取越来越高级的特征。这些特征在一个方向上通过这些 RBM 层前向传递，在顶层产生更精细的特征。

初始化前馈网络。之后使用这些特征层作为由传统反向传播驱动的前馈神经网络的初始权重。

这些初始化值有助于训练算法将传统神经网络的参数引导到更好的参数搜索空间区域。这个阶段被称为 DBN 的**微调阶段**。

2. 用前馈多层神经网络微调DBN

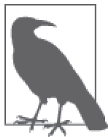
在 DBN 的微调阶段，使用普通的反向传播，以较低的学习率进行“温和”的反向传播。预训练阶段被视作以无监督方式对原始数据参数空间的一般搜索。相比之下，微调阶段针对的是我们实际关心的任务（比如分类），特化网络及其特征。

温和的反向传播。RBM 的预训练阶段从数据中学习高级特征，其被用作前馈网络良好的初始值。使用这些权重，并对它们稍作调整，以找到适合最终神经网络模型的良好的值。

输出层。深度网络通常的目标是学习一组特征，其第一层学习如何重建原始数据集，随后的层学习如何重建前一层激活值的概率分布。神经网络的输出层与总体目标相关联。它通常使用逻辑回归，最终层输入的数量等于特征的数量，输出的数量等于类别的数量。

3. DBN的当前状况

本书没有像对其他网络架构那样详细介绍 DBN。这是因为 CNN 已经统治了图像建模领域，因此我们选择重点介绍 CNN，详见下一节。



DBN 在深度学习兴起中的作用

虽然本书没有多介绍 DBN，但该网络在深度学习的兴起中起到了重要作用。多伦多大学的 Geoff Hinton 团队长期坚持在图像建模领域推广技术，并取得了巨大进展。DBN 在深度网络的演化中起着重要作用，这一点值得注意。

4.1.2 GAN

还有一个需要了解的网络是 GAN。GAN 已被证明非常擅长基于其他训练图像合成新的图像。可以扩展这个概念，对其他领域建模，例如：

- 音频
- 视频
- 从文本描述生成图像

GAN 是使用无监督学习并行训练两个模型的网络的一个例子。GAN（和一般的生成模型）的一个重要特性是相对于正在训练的网络的数据量，它们使用的参数数量明显少于普通网络。网络被迫高效地代表训练数据，使得它能更高效地生成与训练数据类似的数据。

1. 训练生成模型、无监督学习和GAN

如果有大量训练图像材料（比如 ImageNet 数据集），我们可以构建一张输出图像（相对于分类）的生成神经网络。这些生成的输出图像被当作来自模型的样本。GAN 中的生成模型生成这样的图像，而辅助“鉴别器”网络尝试对这些生成的图像进行分类。

辅助鉴别器网络尝试将输出图像分类为真实图像或合成图像。在训练 GAN 时，需要更新参数，以便网络基于训练数据生成更可信的输出图像，目标是使图像真实到足以骗过鉴别器网络，使其不能区分真实输入数据和合成输入数据之间的差异。

一个能够体现 GAN 模型高效的例子是，在对 ImageNet 这样的数据集建模时，它通常具有约 1 亿个参数。在训练过程中，像 ImageNet（200GB）这样的输入数据集会产生接近 100MB 的参数。这个学习过程试图找到最高效的方法来表示数据的特征，如相似的像素组、边缘和其他模式（4.2 节将详细介绍）。

鉴别器网络。对图像建模时，鉴别器网络通常是标准的 CNN。以辅助神经网络为鉴别器网络，可以使 GAN 以无监督的方式并行训练两个网络。这些鉴别器网络以图像为输入，然后输出分类。

鉴别器网络的输出相对于合成输入数据的梯度，指出该如何略微改变合成数据以使其更逼真。

生成网络。GAN 中的生成网络生成一种被称为**反卷积层**的特殊类型层的数据（或图像）（请阅读以下专栏中关于反卷积网络和层的更多内容）。

训练期间，在两个网络上使用反向传播更新生成网络的参数，以生成更逼真的输出图像。这里的目标是将生成网络的参数更新到足以"骗过"鉴别器网络的程度，因为与训练数据的真实图像相比，输出是如此逼真。

反卷积网络

此类网络由纽约大学的 Matthew Zeiler 和 Rob Fergus 在论文"Visualizing and Understanding Convolutional Neural Network"中作为 ZF 网络的一部分而开发。一个反卷积网络有助于检查不同的特征激活值及其与输入空间的关系（见图 4-5）。

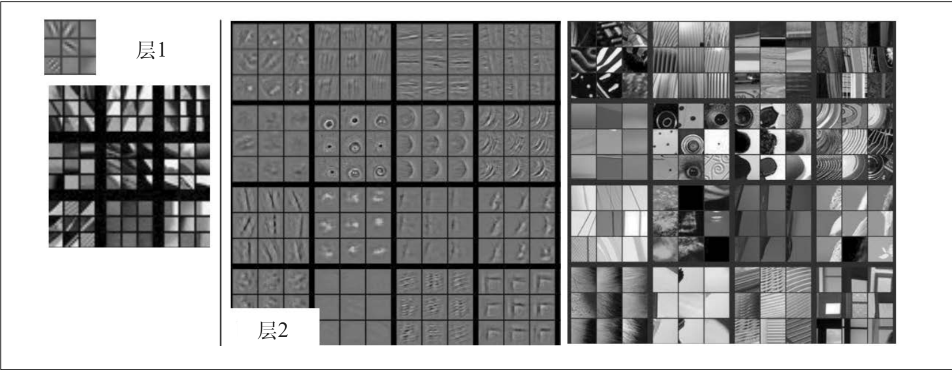


图 4-5：反卷积层可视化

反卷积网络的反卷积层对图像建模时将特征映射到像素，如图 4-5 所示，这与普通的卷积层相反。这一特点使得神经网络能够生成图像的输出。反卷积网络使用无监督和逐层的训练方式，类似于 DBN。该网络有多个叠层反卷积层，其中每一层基于前一层的输入进行训练。它将信息通过层的输出当作该层输入的稀疏表示。

2. 建立生成模型与深度卷积生成对抗网络

GAN 的一个变体是**深度卷积生成对抗网络**（DCGAN）。图 4-6 展示了由 DCGAN 生成的卧室图像。



图 4-6：由 DCGAN 网络生成的卧室图像

这个网络采用随机数（来自均匀分布），并以网络模型生成的图像为输出。随着输入随机数的变化，DCGAN 会生成不同类型的图像。

3. 条件GAN

条件 GAN 也可以使用类别的标签信息，使它们有条件地生成特定类别的数据。

4. GAN与变分自动编码器的比较

GAN 致力于将训练记录归类为模型分布或真实分布。当鉴别器模型预测两个分布之间有差异时，生成器网络调整其参数。最终，生成器收敛于重新生成真实数据分布的参数，并且鉴别器无法检测到该差异。

我们使用变分自动编码器（VAE）建立相同的问题，使用概率图形模型以无监督的方式重建输入，如同第 3 章所讲的。VAE 旨在降低数据对数似然性的下限，使生成的图像看起来更加真实。

GAN 和 VAE 另一个有趣的区别是图像生成的方式。使用基本的 GAN，图像是用任意代码生成的，无法生成具有某种特征的图片。相比之下，VAE 有一个特定的编码 / 解码方案，可以比较生成的图像与原始图像，其连带效果是我们能够为生成特定类型的图像编码。



生成模型的问题

有时生成的输出图像中会出现不同类型的噪声。VAE 生成图像的缺点是，由于其生成图像的方式，图像有时会稍微模糊。GAN 生成的图像倾向于捕获输入数据的样式，但有时不以连贯的方式构成场景（例如可以生成狗的图像，但看起来却不太像狗）。

4.2 CNN

CNN 的目标是通过卷积学习数据中的高级特征。它非常适合图像的目标识别和常年举办的顶级图像分类竞赛。它可以识别脸部、人、街道标志、鸭嘴兽和许多其他类型的视觉数据。CNN 在光学字符识别之上实现了文本分析，也可以用于把单词作为独立的文本单元进行分析。它还擅长分析声音。

CNN 能够高效地识别图像是世人认识到深度学习能力的主要原因之一。如图 4-7 所示，CNN 擅长从原始图像数据建立位置和（一些）旋转不变特征。

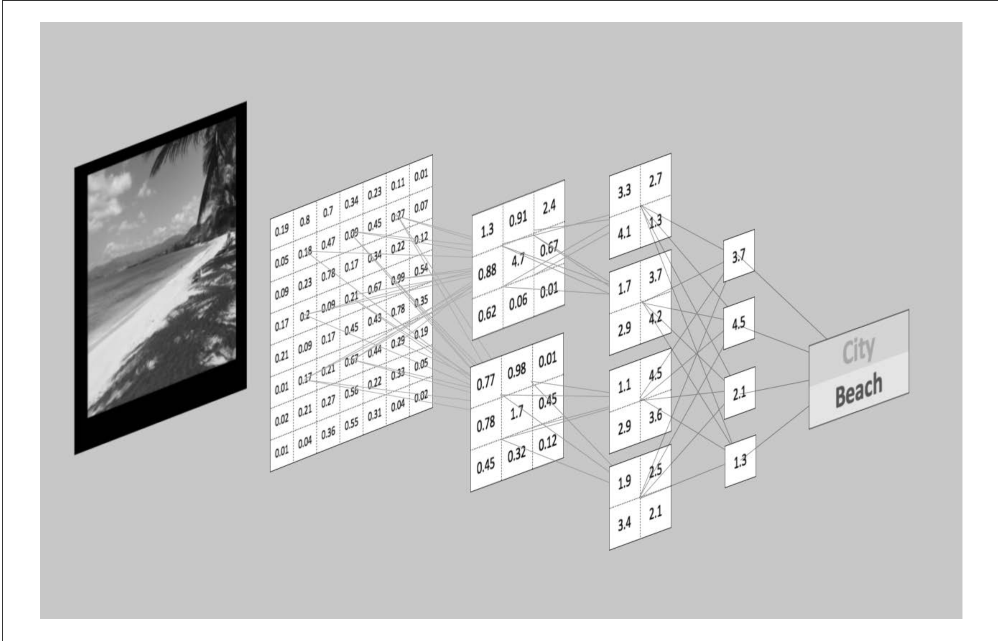


图 4-7：CNN 与计算机视觉

CNN 正推动着机器视觉的重大进步，机器视觉在自动驾驶汽车、机器人、无人机和视力受损的治疗方面有明显的应用。



CNN 与数据结构

当输入数据具有某种结构时，CNN 往往最有用。例如那些具有特定重复模式集的图像和音频数据，彼此相邻的输入值在空间上相关联。相反，从关系数据库管理系统（RDBMS）导出的列式数据在空间上往往没有结构关系。彼此相邻的列只是恰好在数据库导出的物化视图中彼此相邻而已。

CNN 还用于其他任务，例如自然语言翻译 / 生成和情感分析。卷积是帮助我们基于信号构建更强健的特征空间的强大概念。

4.2.1 生物学启发

CNN 受到了动物的视觉皮层的启发。视觉皮层中的细胞对输入的小的子区域敏感，我们称之为**视野**或**感受野**。这些较小的子区域拼接在一起布满整个视野。这些细胞非常适合利用大脑处理的图像类型中的强空间局部相关性，并在输入空间中充当局部过滤器。大脑的这一区域有两种细胞，简单的细胞在检测类似于边缘的图案时激活，而更复杂的细胞在拥有较大的感受野且不随图案位置变化时激活。

4.2.2 思路

前馈多层神经网络以输入为单一的一维向量，用一个或多个隐藏层（全连接）转换数据，之后网络从输出层给出结果。对于传统的多层神经网络和图像数据，出现的问题是这些网络对输入为图像数据的场景伸缩性不好。例如对 CIFAR-10 数据集建模（参见之后的专栏），训练的图像是只有 32 个像素宽、32 个像素高、3 个通道的 RGB 信息。然而这使得第一个隐藏层为每个神经元创建了 3072 个权重，并且该隐藏层中往往不止一个神经元。在许多情况下，多层神经网络中会有多个隐藏层，这也会增加权重。

CIFAR-10 数据集

CIFAR-10 数据集是由 Alex Krizhevsky、Vinod Nair 和 Geoffrey Hinton 编排的著名的**图像分类基准数据集**。其中有 60 000 张彩色图像，包含 10 个不同类别，每个类别包含 6000 张图像。每张图像都为 32 像素 ×32 像素。数据集中有 50 000 张训练图像和 10 000 张测试图像。图 4-8 展示了数据集中图像的类别。



图 4-8：CIFAR-10 数据集

这些类别互不重叠，意味着卡车的图像只包含卡车的图像。数据集的大小约为 170MB。

通常，图像可以轻松达到 300 个像素宽、300 个像素高、3 个 RGB 信息通道，这将导致每个隐藏层的神经元具有 270 000 个连接权重。这显示了全连接的多层网络在对图像数据建模时连接的数量增长速度有多快。我们可以利用这些图像数据的结构来改变神经网络架构。使用 CNN 可将神经元按下面的三维结构排列：

- 宽度
- 高度
- 深度

输入的这些属性与图像结构相对应。

- 图像的像素宽度
- 图像的像素高度
- RGB 通道为深度

可以把这种结构看作三维的神经元空间。CNN 从以前的前馈变体发展而来的一个重要特性是它利用新的类型的层实现计算效率，很快我们将详细介绍这一特性。下面从高的层面了解 CNN 的常见架构。

4.2.3 CNN架构概要

CNN 将来自输入层的输入数据，通过全连接层，转换为由输出层给出的一组类别分数。CNN 架构有很多变体，但都是基于层模式，如图 4-9 所示。

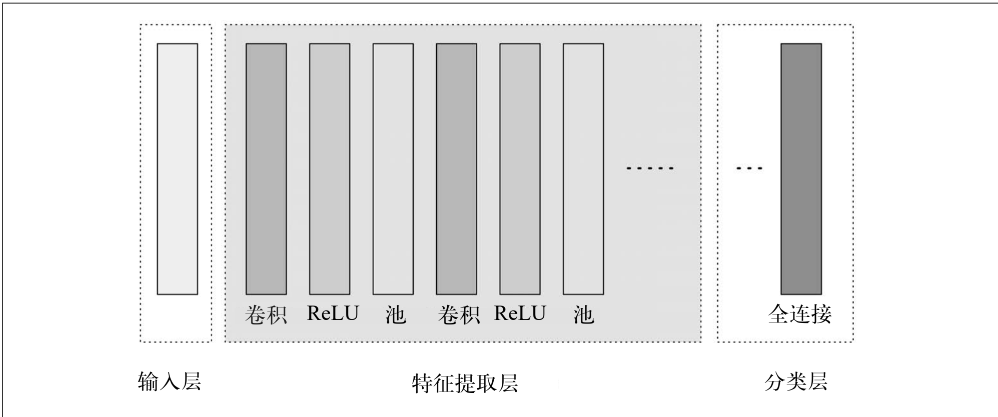


图 4-9：在高的层面上抽象的常见 CNN 架构

图 4-9 描述了三个主要类别：

(1) 输入层；
(2) 特征提取（学习）层；
(3) 分类层。

输入层通常接受图像大小（宽度 × 高度）的空间形式的三维输入，并且具有表示颜色通道的深度（通常是 3 个 RGB 颜色通道）。



将小批量作为第四维度

将样本组合为训练用的小批量，最终会得到四个维度——最后一个维度用于索引小批量中的样本。因此，在 DL4J 中，图像训练数据的数组有四个维度，而不只是三个。

特征提取层处理序列的常见且重复的模式如下所示。

(1) 卷积层。
　　为和其他文献保持一致，这里将 ReLU 激活函数作为图中的一个图层。
(2) 池化层。

这些层在图像中发现大量特征，并逐步构造高级特征。这与不采取传统的人工方式，而是自动学习特征这一深度学习进展中的主题直接相呼应。

最后是分类层，其中一个或多个全连接的层用于获取高级特征，并生成类别概率或分数。顾名思义，这些层与前一层中的所有神经元全连接。这些层通常生成大小为 $[b \times N]$ 的二维输出，其中 b 是小批量中的样本数量，N 是我们想去评分的类别的数量。

1. 神经元空间排列

回想一下在传统多层神经网络中层是全连接的，层中的每个神经元都与下一层的每个神经元连接。在 CNN 层中神经元在三个维度上被排列以匹配输入空间。在这里，深度意味着激活空间的第三维，而不是多层神经网络中层的数量。

2. 层间连接的发展

卷积架构的另一个变化是连接层的方式。某一层的神经元只与前一层一小部分区域的神经元连接。CNN 同传统的多层网络一样，保留了层级架构，但有不同类型的层。每一层将来自前一层的三维输入空间转换为用于神经元激活层的三维输出空间，这些激活层有一些可能有参数也可能没有参数的可微函数，如图 4-10 所示。

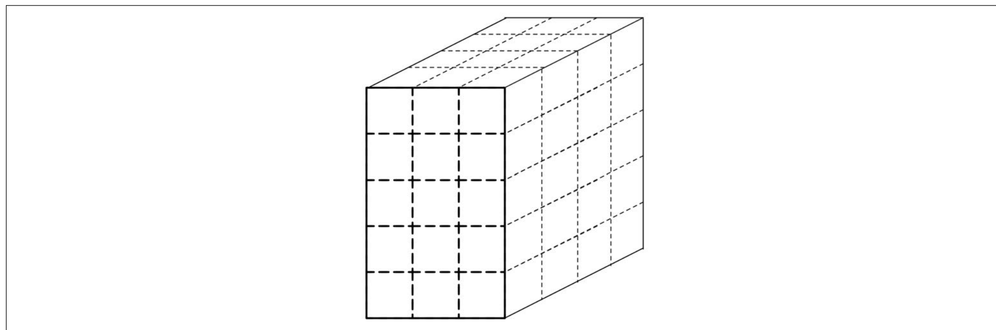


图 4-10：输入层三维空间

4.2.4　输入层

输入层加载和存储图像原始输入数据，以便在网络中处理。输入数据指定了宽度、高度和通道的数量。通道数量通常为 3，对应每个像素的 RGB 值。

4.2.5　卷积层

卷积层被视为 CNN 架构的核心构造块。如图 4-11 所示，卷积层使用来自前一层的局部连接神经元转换输入数据。该层将计算输入层中神经元的区域与输出层中神经元局部连接的权重之间的点积。

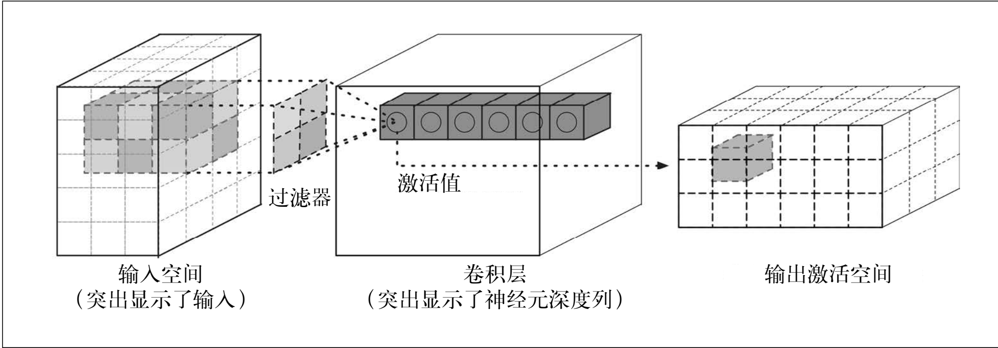


图 4-11：具有输入和输出空间的卷积层

输出通常具有相同的空间维度（或更小的空间维度），但有时会增加输出的第三维度（深度）中元素的数量。接下来仔细研究这些层中的一个关键概念：**卷积**。

1. 卷积

我们将卷积定义为描述合并两组信息的规则的数学运算，在物理学和数学中都很重要。它使用傅里叶变换，在空间 / 时间域和频域之间定义了一个桥梁。它接收输入，应用卷积核，并给出一个特征映射作为输出。

图 4-12 所示的卷积运算被称为 CNN 的**特征检测器**。卷积的输入可以是原始数据或由另一个卷积输出的特征映射。通常把它解释为过滤器，由核过滤输入数据以获得某些类型的信息。例如边缘核只允许来自图像边缘的信息通过。

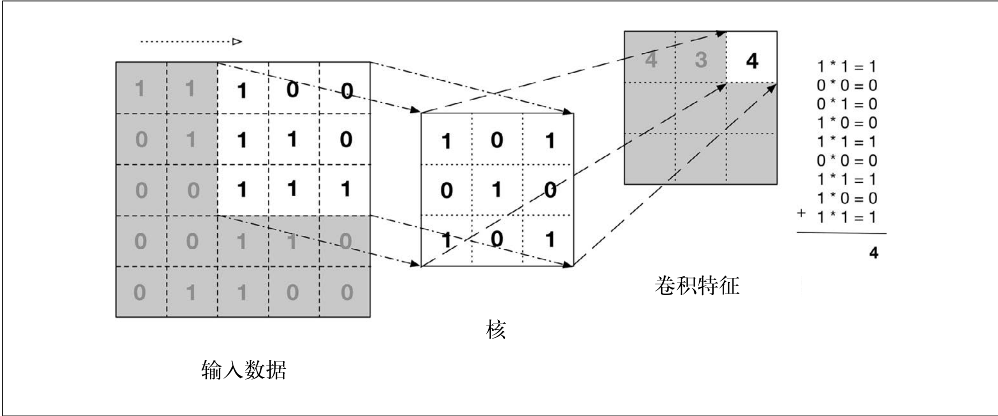


图 4-12：卷积运算

这张图展示了如何在输入数据上滑动核以生成卷积的特征（输出）数据。在每个步骤中，核乘以在其边界内的输入数据值，在输出特征映射中创建一个条目。在实践中，如果检测到在输入中寻找的特征，那么输出会很大。

通常把卷积层中的权重集称为“过滤器”（或核）。该过滤器对输入进行卷积，结果为特征映射（或激活映射）。卷积层对输入数据空间执行转换，这些转换是输入空间中激活值和参数（神经元的权重和偏置）的函数。每个过滤器的激活映射沿深度堆叠在一起，以构建三维输出空间。

卷积层具有用于层和附加超参数的参数。使用梯度下降训练该层中的参数，使类别的分数与训练集中的标签一致。卷积层的主要组件如下所示：

- 过滤器
- 激活映射
- 参数共享
- 层特有的超参数

下面介绍每个组件的细节。

2. 过滤器

卷积层的参数配置层的过滤器集合。过滤器是宽度和高度上比输入空间小的函数。



自然语言处理应用中的过滤器大小

过滤器的大小可以等于输入空间，但通常只在一个维度上，而不会在两个维度上，将 CNN 应用于自然语言处理（NLP）时需要注意。

过滤器（例如卷积）以滑动窗口的方式适应输入空间的宽度和高度，如图 4-12 所示。过滤器也适应输入空间的每个深度。我们通过计算过滤器和输入区域的点积来计算过滤器的输出。



过滤器计数和激活映射

我们将对输入空间应用过滤器的输出称作该过滤器的**激活映射**（也称**特征映射**）。在许多 CNN 图中，经常看到许多小的激活映射。这些映射是如何产生的？这个问题有时令人困惑。

过滤器的数量是每个卷积层的超参数值。这个超参数还控制着从卷积层产生多少激活映射作为到下一层的输入，可以将其看作三维层输出激活空间的第三维（激活映射的数量）。过滤器数量超参数可以自由选择，但设置为某些值会工作得更好。

CNN 的架构被设置为学习过滤器对空间局部输入模式生成最强的激活值。这意味着，只有当模式出现在各自领域的训练数据中时，过滤器才会被学习来激活模式（或特征）。随着在 CNN 中层层深入，会见到能够识别特征的非线性组合的过滤器，并且它们检测到的模式会越来越具有全局性。高性能的卷积架构（稍后将介绍）已经表明网络深度是 CNN 的一个重要因素。

3. 激活映射

回想一下第 1 章开始的内容，如果神经元决定让信息通过，激活值就是一个数值结果。这是激活函数的输入和连接上的权重（用于输入，以及激活函数本身的类型）的函数。当过滤器“激活”时，意味着过滤器允许信息从输入空间传递到输出空间。

在信息经 CNN 前向传递期间，我们在输入的空间维度（宽度、高度）上滑动每个过滤器，这就为特定过滤器生成一个被称为“激活映射”的二维输出。图 4-13 展示了这个激活映射如何与之前介绍的卷积特征的概念相关联。

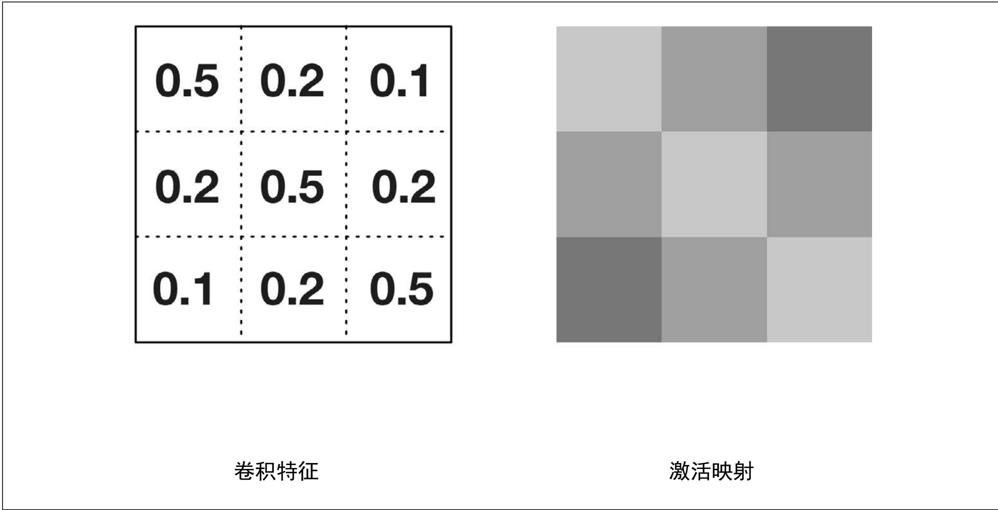


图 4-13：卷积与激活映射

图 4-13 中右侧的激活映射以不同的方式呈现，以说明卷积激活映射在文献中通常的呈现方式。



激活映射

一些文献将激活映射输出称作**特征映射**，但本书称其为“激活映射”。

在输入空间深度切片上滑动过滤器以计算激活映射。我们计算过滤器中条目和输入空间之间的点积。过滤器表示与输入激活值的移动窗口（子集）相乘的权重。网络学习过滤器，当其在特定空间位置看到输入数据中某些类型的特征时激活。

沿输出的深度堆叠这些激活映射，来为卷积层创建三维输出空间，如图 4-14 所示。输出空间会包含那些只观察输入空间的一个小窗口的神经元的输出。

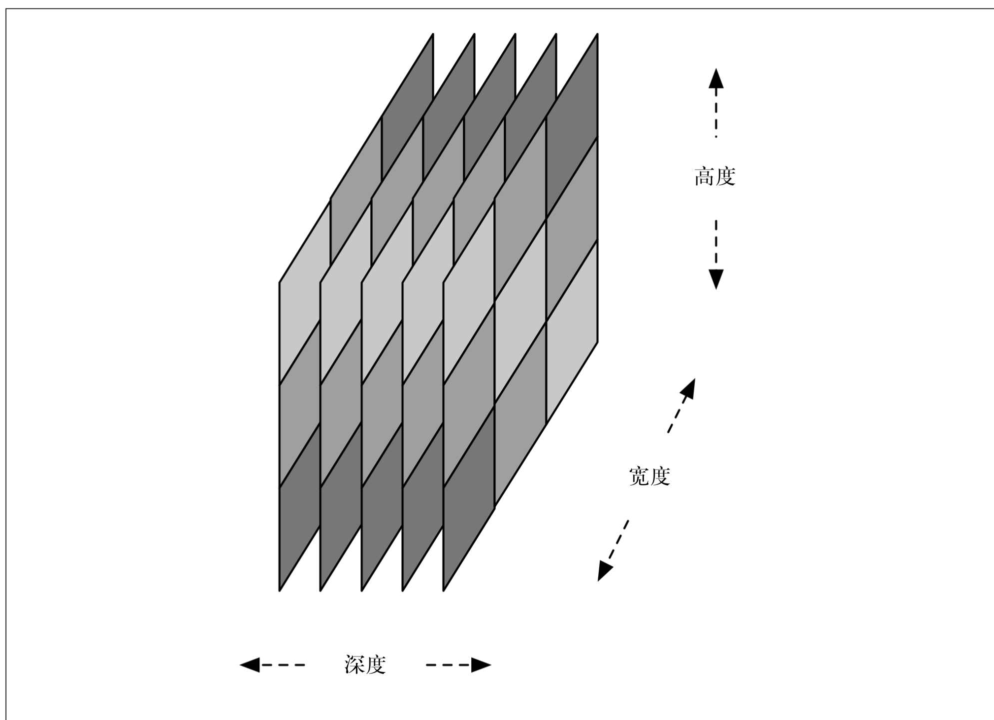


图 4-14：卷积层的激活空间的输出

在某些情况下，这个输出将是同一激活映射中神经元共享参数的结果。生成输出空间的每个神经元只连接到输入空间的局部区域，如图 4-15 所示。

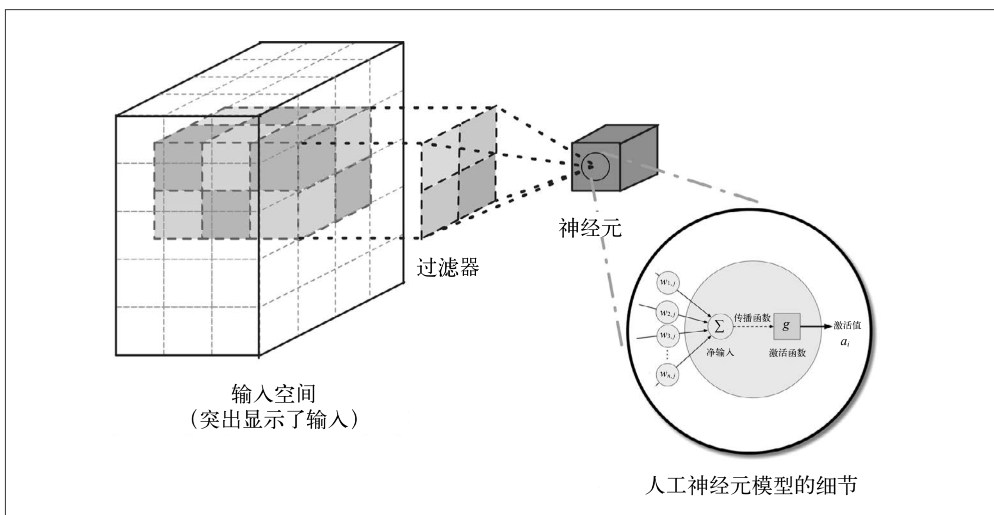


图 4-15：生成激活输出空间

我们使用被称为**感受野**的超参数控制这个过程的局部连通性，它控制过滤器映射的输入空间的宽度和高度。

> **用感受野控制局部连通性**
>
> 层中的神经元沿空间维度（宽度和高度）连接到输入空间的较小区域，但沿深度轴的连接总是等于输入空间的深度，这意味着输入空间的深度总是有充分的连通性。来看一个基于 CIFAR-10 输入数据（RGB 图像）的例子，这个例子在之前的“CIFAR-10 数据集是什么”专栏出现过。
>
> 在这种情况下，输入空间的大小为 $32 \times 32 \times 3$，感受野超参数被设置为 5×5。卷积层中每个神经元都有一个连接到输入空间中 $5 \times 5 \times 3$ 区域的权重。这样卷积层中每个神经元都有 $5 \times 5 \times 3 = 75$ 个权重。
>
> 输入空间的深度为 3，它总是卷积层上权重连通性的深度。神经元连接区域的宽度和高度小于图像，但深度始终与图像保持相同。
>
> 卷积层中连通性可能是局部的，但神经元本身保持不变。我们仍然用非线性函数的输入计算权重的点积。
>
> 现在唯一的区别是神经元只与输入的一个子集相连，而不是像传统的多层神经网络那样与前一层的每个神经元相连。这种连通性是全深度的，但在空间上是局部的。

过滤器定义较小的有界区域，从输入空间生成激活映射。它们通过之前描述的局部连通性，仅与输入空间的子集动态连接。这在保证有质量的特征提取的同时，减少了每层需要训练的参数数量。卷积层使用被称为**参数共享**的技术进一步减少参数数量。

4. 参数共享

CNN 使用参数共享方案控制参数总量，这有助于减少训练时间，因为这将使用更少的资源来学习训练数据集。为了实现 CNN 中的参数共享，首先将单个二维深度切片定义为“深度切片”，然后约定每个深度切片中的神经元使用相同的权重和偏置，这使给定卷积层的参数（或权重）数量显著减少。

当训练的输入图像具有特定的中心结构时，不能利用参数共享。当我们总是期望某个特征出现在特定的地方（处于脸部中央）时，会看到它对脸部的影响。在这种情况下，不使用参数共享。参数共享也是使 CNN 对平移 / 位置不变的技术。

5. 学到的过滤器和渲染器

图 4-16 给出了学到的 96 个大小为 $11 \times 11 \times 3$ 的过滤器的例子。使用参数共享方案时，由于图像的平移不变性，检测的水平边缘在图像的许多位置都有用。这意味着只要在一个位置学习水平边缘，就无须在图像的所有位置学习它作为特征。

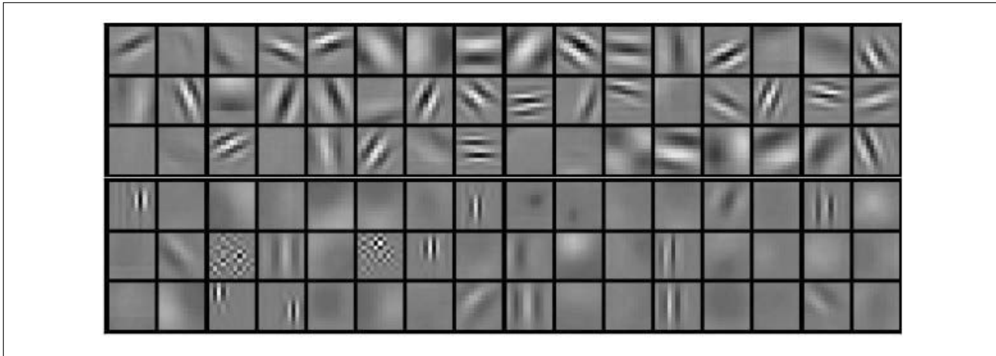


图 4-16：由 Krizhevsky 等提供的学到的过滤器示例（96 个过滤器，11×11×3）

稍稍分解一下，想想二维图像，如果将图像细分为四个部分，神经网络将学习图像的位置不变特征。位置不变性存在是由于网络将数据细分成象限的做法。之后，它一次学习图像的一部分，并将结果池化。这使得神经网络学习到全局的特征，而不是任何整体的局部子集。第 6 章将详细探讨生成过滤器和渲染器的内容。



CNN 特征的旋转不变性

设计上学到的特征是位置不变的，但（通常）不是旋转不变的，不过通过适当的数据增强，可以实现一定程度的旋转不变性。

6. 作为层的ReLU激活函数

ReLU 层在 CNN 中被大量使用。ReLU 层对输入数据阈值应用元素激活函数，例如 `max(0,x)`，在 0 点会得到同样维度的输出作为层的输入。



DL4J、层类型和激活函数

在 DL4J 中，通过神经元激活函数的类型识别层（但这并不总是反映在层的类别名称中）。DL4J 在层中内置了激活函数。Caffe 及其他一些的库使用单独的激活层。

在输入空间上运行该函数将改变像素值，但不会改变输出中输入数据的空间维度。ReLU 层没有参数，也没有额外的超参数。

7. 卷积层超参数

下面是决定空间排列和卷积层输出空间大小的超参数[2]。

- 过滤器（或核）大小（字段大小）
- 输出深度
- 步长
- 零填充

注 2：CNN 中另一个常见的超参数叫作“扩张”。目前 DL4J（0.7 版本）还不支持扩张。



第 7 章将介绍更多关于卷积层大小的内容

本节解释这些超参数的工作方式，第 7 章将介绍 CNN 层的调优机制。

过滤器大小。相对于过滤器尺寸的宽度和高度，每个过滤器在空间上很小。例如，第一个卷积层可能具有 5×5×3 大小的过滤器。这意味着，假设输入是 3 通道 RGB 颜色的图像，那么过滤器是 5 像素宽、5 像素高，3 代表颜色通道。

输出深度。可以手动选择输出空间的深度。深度超参数控制卷积层中连接到输入空间相同区域的神经元数量。



边缘与激活

沿着深度的不同神经元学会了在受到创建的输入数据（例如颜色或边缘）的刺激时激活。

我们将一组接受输入空间同一区域的神经元视为深度列。

步长。步长用于配置滑动过滤器窗口在每次应用过滤函数时移动的距离。每次将过滤函数应用到输入列时，会在输出空间中创建一个新的深度列。较小的步长设置（例如，1 表示指定仅仅一个单元的步长）将在输出空间中分配更多的深度列。这也会在列之间生成重叠度更高的感受野，从而导致输出空间更大。当指定较大的步长时，情况正好相反。这些更大的步长使空间中的堆叠较少、输出空间更小。

零填充。最后一个超参数是零填充，用它可以控制输出空间的大小。想保持输出空间中输入空间大小时可以这样做。

8. 批量规范化和层

为了加速 CNN 的训练，可以在每个批量中规范化前一层的激活值。这种技术应用一个转换，使平均激活值保持在接近 0.0 的同时，使激活值的标准差保持在 1.0 附近。

CNN 的批量规范化表明，规范化网络架构可以加快训练速度。通过对每个训练小批量的输入记录应用规范化，可以使用更高的学习率。批量规范化还降低了训练对权重初始化的敏感性，并且充当了正则化器（减少了对其他类型正则化的需求）。批量规范化也已应用于 LSTM 网络，它是另一种本章稍后将讨论的深度网络。

4.2.6 池化层

池化层通常被插入在连续的卷积层之间。使用池化层跟随卷积层来逐步减小数据表示的空间大小（宽度和高度）。池化层在网络上逐步减少数据表示，并有助于控制过拟合，它在输入的每个深度切片上独立执行。



常用的下采样操作

最常用的下采样操作是取最大值的 max **操作**。次常用的操作是**平均池化**。

池化层通过 `max()` 操作调整输入数据的空间（宽度、高度）大小。这个操作被称为**最大池化**。使用 2×2 大小的过滤器，`max()` 操作在过滤器区域中取四个最大的数字，该操作不影响深度。

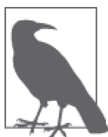
池化层使用过滤器在输入空间执行下采样过程。这些层沿输入数据的空间维度执行下采样操作，这意味着如果输入图像是 32 像素宽、32 像素高，那么输出图像的宽度和高度将更小（如 16 像素宽、16 像素高）。池化层最常见的设置是应用 2×2 过滤器，步长为 2。这将在一半的空间维度（宽度和高度）上对输入空间的每个深度切片进行下采样。这种下采样操作将导致 75% 的激活被丢弃。

池化层没有参数，却有额外的超参数。该层不涉及参数，因为它是计算输入空间的固定函数。将零填充用于池化层不太常见。

4.2.7　全连接层

使用该层计算用作网络输出类别的分数（例如网络末端的输出层）。输出空间的维度是 $[1 \times 1 \times N]$，其中 N 是正在评估的输出类别的数量。对于之前讨论的 CIFAR 数据集，由于数据集中有 10 个类别，N 为 10。该层所有神经元和前一层每个神经元之间都有连接。

全连接层有用于层和超参数的一般参数。全连接层对输入数据量执行转换，该转换是输入空间中激活和参数（神经元的权重和偏置）的函数。



多个全连接层

有一些 CNN 架构在网络的末端使用多个全连接层。例如 AlexNet，它有两个全连接层，最后一层是 softmax 层。

4.2.8　CNN的其他应用

除了普通的二维图像数据外，CNN 还用于三维数据集。下面是一些其他应用的例子。

- MRI 数据
- 三维形状数据
- 图数据
- NLP 应用

CNN 的位置不变特性在这些领域已被证明是有用的，因为不限制手动编码的特征在特征向量的某些“位置”出现。

4.2.9 CNN列表

下面是一些更流行的 CNN 结构的列表。

- LeNet
  - 最早成功的 CNN 架构之一；
  - 由 Yann Lecun 开发；
  - 最初用于读取图像中的数字。
- AlexNet
  - 使 CNN 在计算机视觉领域流行；
  - 由 Alex Krizhevsky、Ilya Sutskever 和 Geoff Hinton 开发；
  - 2012 年度 ILSVRC 竞赛冠军。
- ZF Net
  - 2013 年度 ILSVRC 竞赛冠军；
  - 由 Matthew Zeiler 和 Rob Fergus 开发；
  - 引入了反卷积网络的可视化概念。
- GoogLeNet
  - 2014 年度 ILSVRC 竞赛冠军；
  - 由 Christian Szegedy 和他在谷歌的团队开发；
  - 代号“Inception”，有 22 层。
- VGGNet
  - 2014 年度 ILSVRC 竞赛亚军；
  - 由 Karen Simonyan 和 Andrew Zisserman 开发；
  - 证明网络深度是获得良好表现的关键因素。
- ResNet
  - 在非常深的网络上训练（最多 1200 层）；
  - 2015 年度 ILSVRC 竞赛分类任务第一名。

4.2.10 小结

由于对图像数据专门特征提取的需求，CNN 得以发展，无论层在哪列“漫游”，都能很好地找到特征。了解过卷积层、池化层和常规全连接层是如何协同工作来对图像进行分类的，下面转向一个擅长对时间域建模的神经网络架构：RNN。

4.3 RNN

RNN 是前馈神经网络家族的一员。不同于其他前馈网络，它在时间步上发送信息的能力不同。下面是著名研究员 Juergen Schmidhuber 对 RNN 的一个有趣的解释。

> （RNN）允许并行和顺序计算，原则上可以计算传统计算机计算的任何内容。然而，与传统计算机不同，RNN 类似于人脑，人脑是一个由相连神经元组成的大型反馈网络，神经元终其一生以某种方式将感觉输入流转换成一系列有用的运动输出。大脑是一个伟大的榜样，它能解决当前机器无法解决的许多问题。

在历史上，这些网络很难训练，但最近研究的进展（调优、网络架构、并行计算和 GPU）使它们对实践者更友好了。

RNN 从输入向量序列中提取每个向量，并且一次对一个向量建模。这使得网络在输入向量窗口上对每个输入向量建模的同时，还能保持状态。对时间维度建模是 RNN 的标志性特征。

4.3.1 时间维度建模

RNN 被视作图灵完备的，可以模拟任意（带权重的）程序。如果把神经网络看作对函数的优化，可以把 RNN 看作“对程序的优化”。RNN 非常适合对由涉及值之间时间依赖的向量组成的输入或输出的函数建模。RNN 通过在网络中创建循环（即名称中“循环”部分的由来）来对数据的时间维度建模。

1. 迷失在时间中

许多分类工具（支持向量机、逻辑回归和常规前馈网络）在没有对时间维度建模且假设是独立的情况下，已被成功应用。这些工具的其他变体通过对输入的滑动窗口（例如，上一个、当前的和下一个输入一起作为单个输入向量）建模，来捕获时间动态。

这些工具的缺点是假设模型输入之间的时间连接是独立的，那么模型就不能捕获长期的时间依赖。滑动窗口技术的窗口宽度有限，不能捕获大于固定窗口的效应。例如对会话工作方式建模，并且让机器理解在会话随着时间推移时如何以一致的方式回复。一个训练有素的 RNN 可以完成艾伦·图灵著名的**图灵测试**，骗过人类，让人以为在和真人说话。

2. 连接中的时间反馈和循环

RNN 在连接中可以具有环路，这使得它能够在时间序列、语言、音频和文本等领域对时间行为建模时提高准确度。



输出层到隐藏层的连接的说明

在实践中，这种连接方案较少，DL4J 也不使用这个方案。但本书认为了解这种变体很重要，每个循环层中常有从一个时间步到下一个时间步的神经元连接。

这些域中的数据本质上是有序的、上下文敏感的，后面的值取决于先前的值。RNN 的线路能够捕获这些时间效应，进行反馈。RNN 架构常用于时间序列域的应用。

RNN 包括一个反馈环路，用于从不同长度的序列中学习。RNN 包含用于时间步之间连接的额外参数矩阵，这些时间步被使用 / 训练以捕获数据中的时间关系。

RNN 通过训练生成序列，其中每个时间步的输出基于当前输入和在所有先前时间步的输入。普通的 RNN 通过一种被称为**基于时间的反向传播**（BPTT）的算法计算梯度。本章稍后详细介绍 BPTT。

3. 序列和时间序列数据

在模型需要输出向量序列的行业的许多问题领域会见到序列数据。

- 图像字幕

- 语音合成
- 音乐生成
- 玩电子游戏
- 语言建模
- 字符级文本生成模型

在其他领域，需要输入向量的序列有：

- 时间序列预测
- 视频分析
- 音乐信息检索

同时需要输入和输出向量序列的领域有：

- 自然语言翻译
- 参与对话
- 机器人控制

与其他深度网络相比，RNN 可以对以下类型的输入建模（非固定输入）。

- 非固定计算步骤。
- 非固定输出大小。
- 它可以操作向量序列，例如视频帧。

RNN 的一个重要特点是它以独特的方式处理输入和输出。

4. 理解模型的输入和输出

传统的机器学习基于单个固定大小的输入向量进行操作。传统建模活动中，固定输入大小和固定输出大小的从输入到输出的关系很常见。

这通常是用于构建图像分类或列式数据分类器模型的模式。

RNN 将输入动态改变为包括多个输入向量，每个时间步具有一个向量，并且每个向量可以有多列。下面的列表是 RNN 如何操作输入和输出向量序列的例子。

- 一对多：序列输出。例如为图像配字幕的操作以图像为输入，输出单词序列。
- 多对一：序列输入。例如对给定句子的输入进行情感分析。
- 多对多：例如视频分类，标记每一帧。

研究过输入和输出数据的不同类型，下面介绍这些输入数据的表示方法。

4.3.2 三维空间输入

RNN 的输入比标准机器学习模型输入涉及更多维度，它在概念上与 CNN 相似。输入有以下三个维度：

(1) 小批量大小；
(2) 每个时间步向量的列数；
(3) 时间步的数量。

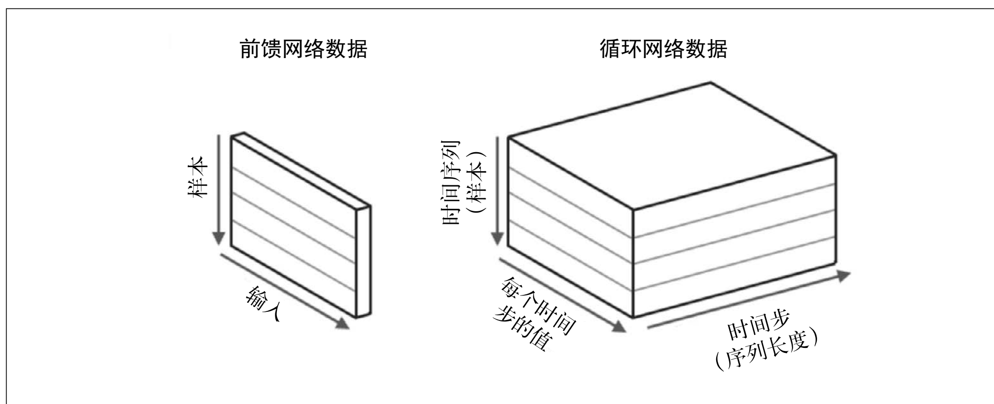
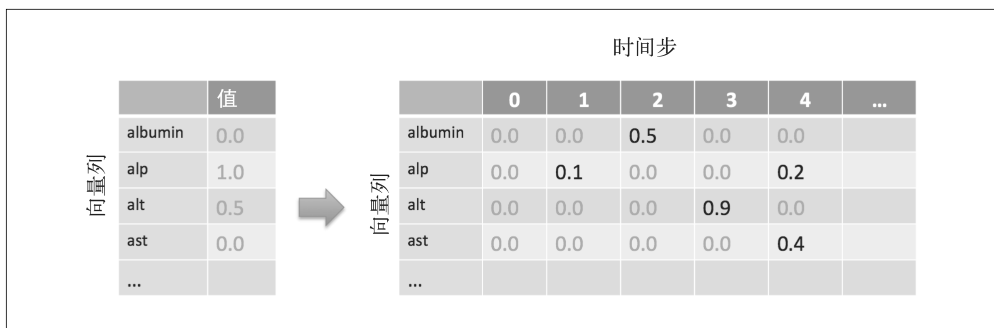


图 4-17：普通输入向量与 RNN 输入的比较

小批量的大小是我们想建模的每个批量输入记录的数量（单个源实体的时间序列点集合）。列的数量与普通输入向量中找到的传统特征列的数量匹配。时间步的数量是随着时间变化而产生的输入向量的变化，这是输入数据的时间序列方面。基于前面所讲的术语，时间步的数量大于 1 意味着输入和输出架构是多维的。

不均匀的时间序列与掩码

之前介绍了 RNN 的输入，除了输入向量的特征外，还有时间步的概念。图 4-18 提供了可视化表示。



	值
albumin	0.0
alp	1.0
alt	0.5
ast	0.0
...	

	0	1	2	3	4	...
albumin	0.0	0.0	0.5	0.0	0.0	
alp	0.0	0.1	0.0	0.0	0.2	
alt	0.0	0.0	0.0	0.9	0.0	
ast	0.0	0.0	0.0	0.0	0.4	
...						

图 4-18：RNN 输入的时间步

每个列值可能不会在每个时间步上都出现，特别是将描述性数据（例如静态数据库表的列）和时间序列数据（例如 ICU 患者每分钟心跳的测量）混合的情况。对于“参差不齐”的时间步的值，需要使用**掩码**来让 DL4J 知道真实数据在向量中的位置。我们通过为掩码提供额外的矩阵来完成此操作，该矩阵表示输入数据的时间步具有至少一列，如图 4-19 所示。

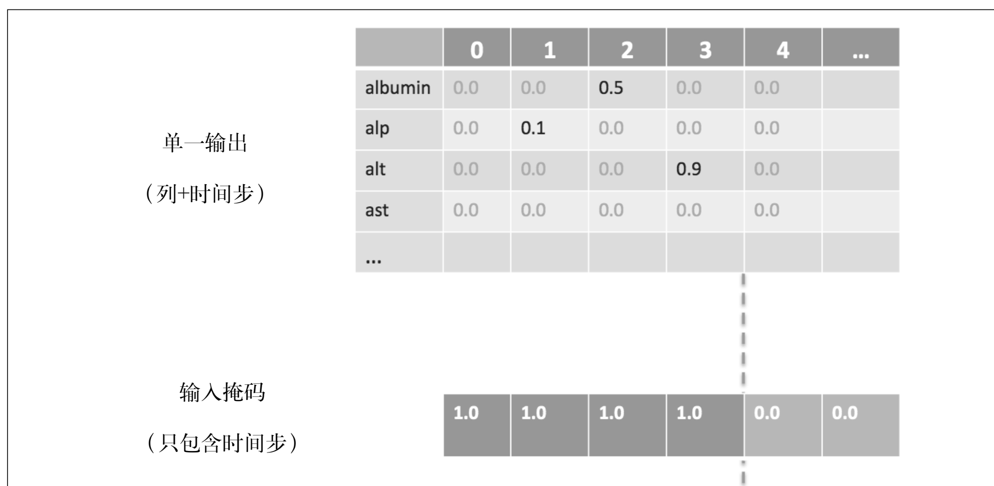


图 4-19：特定的时间步掩码

第 3 章提供了一个如何设置这些掩码的代码示例。

4.3.3 为什么不是马尔可夫模型

在处理模型中的时间维度时，可以把马尔可夫模型看作一种选择，它是另一类广泛用于对序列建模的机器学习模型。其局限在于，随着上下文窗口大小的增加，这些模型最终在对长期依赖建模时变得在计算上不实用。

RNN（连接模型）优于马尔可夫模型（和其他时间窗口有限的模型），因为它可以捕获输入数据中的长期依赖。它能够实现这一点，是因为其隐藏状态从任意长的上下文窗口捕获信息，不像其他技术那样受到限制。此外，它可以建模的状态数量由节点的隐藏层表示，并且这些状态随层中节点数量呈指数式增长。这使得 RNN 在捕获许多输入向量上时间维度的大量相关信息方面异常优秀。

RNN、隐藏层和状态数量

如果输入仅是二元值 (0, 1)，那么网络可以表示 2^N 个状态，其中 N 是隐藏层中的节点数。如果输出是 64 位实数，这些节点的单个隐藏层可以表示 2^{64^N} 个不同状态。

这些网络的训练能力仅随隐藏节点数呈二次增长，而网络的表达能力则随隐藏节点数呈指数式增长。

4.3.4 常见的RNN架构

RNN 是前馈神经网络的超集，但增加了循环连接的概念。这些连接（或循环边）跨相邻时间步（如前一时间步），给模型带来时间的概念。传统的连接不包含 RNN 中的环，而循环连接可以形成环，原始的神经元在未来的时间步可以连接回其本身。

RNN的结构与时间步

在循环网络中发送输入的每个时间步中，沿着循环边缘接收输入的节点从当前输入向量和网络先前状态的隐藏节点接收输入的激活值。

对于给定的时间步，从隐藏状态计算输出。前一时间步的输入向量可以通过循环连接影响当前时间步的输出。

可以把这些特殊的循环神经元层串联起来，建立更好的模型。将前一层的输出与下一层的输入连接，就像前馈多层神经网络的连接方式一样。

> **梯度消失问题**
>
> RNN 有一个已知的“梯度消失问题”。当梯度变得太大或太小时，这个问题就会出现，使得难以对输入数据集结构的长期依赖性（10 个时间步或更多）建模。解决这一问题最有效的方法是使用 DL4J 提供的 RNN 的 LSTM 变体。

4.3.5 LSTM网络

LSTM 网络是 RNN 最常用的变体，由 Hochreiter 和 Schmidhuber 于 1997 年发明。

LSTM 的关键组件是记忆单元和门（包括遗忘门，也是输入门）。记忆单元的内容由输入门和遗忘门调整。假设这两个门都关闭，那么记忆单元的内容在一个时间步和下一个时间步之间保持不变。门结构允许信息跨多个时间步依然得以保留，而且也允许梯度跨多个时间步传递。这使得 LSTM 模型克服了大多数 RNN 模型都会发生的梯度消失问题。

1. LSTM网络的特性

LSTM 有以下良好的特性。

- 更好的更新方程
- 更好的反向传播

下面是 LSTM 的一些用途的例子。

- 生成句子（例如字符级语言模型）
- 时间序列分类
- 语音识别
- 手写识别
- 复调音乐建模

近年来，LSTM 和双向 RNN（BRNN）架构在以下任务上显示了行业领先水准。

- 图像字幕
- 语言翻译
- 手写识别

LSTM 网络由许多相连的 LSTM 单元组成，在学习过程中表现良好。



关于 LSTM 训练复杂度的说明

前向和反向传递操作的计算复杂度随输入序列的时间步的数量线性增减。

下面几节将介绍 LSTM 的架构和组件。

2. LSTM网络的架构

为了更好地理解 LSTM 网络中单元与层之间复杂的连接，先回顾之前提到的一些概念。

前面的章节介绍了前馈多层神经网络的概念，如图 4-20 所示。

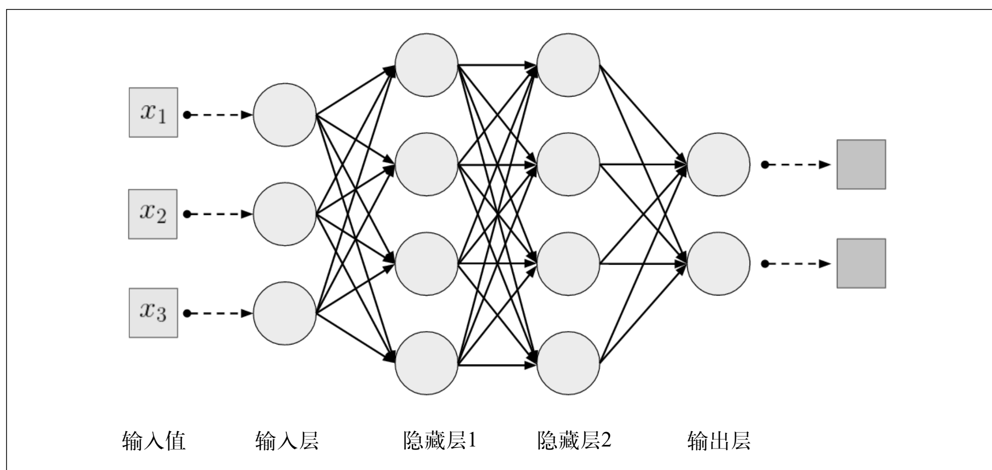


图 4-20：前馈多层神经网络架构

如果将图 4-20 所示的网络的每层作为单个节点以扁平方式表示出来，效果如图 4-21 所示。

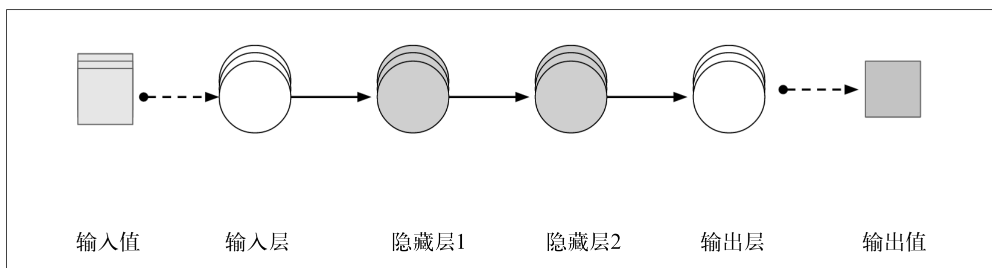


图 4-21：前馈多层网络可视化

RNN 引入了一种连接方式，以隐藏层神经元的输出为输入与同一隐藏层的神经元连接。通过这种循环连接，可以从前一时间步获取输入，作为传入神经元信息的一部分。

图 4-22 是将图 4-21 所示的网络扁平化展开，可以更容易地看清楚 LSTM 中的循环连接。

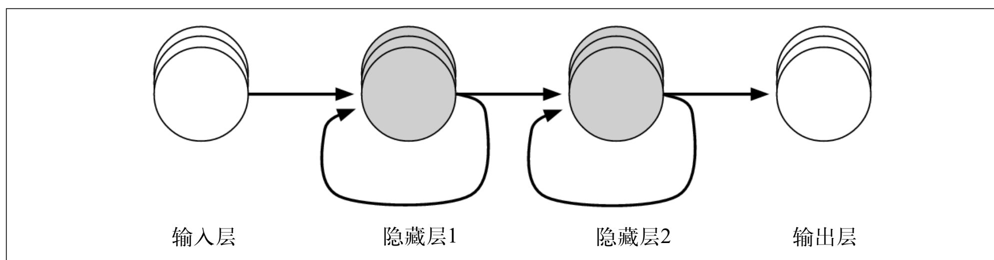


图 4-22：在隐藏层节点上显示循环连接

图 4-23 通过对图 4-22 的网络图的“展开”有助于更直观地理解这一点，图中显示了信息是如何以前馈方式在网络中“流动”并“跨越时间”的。

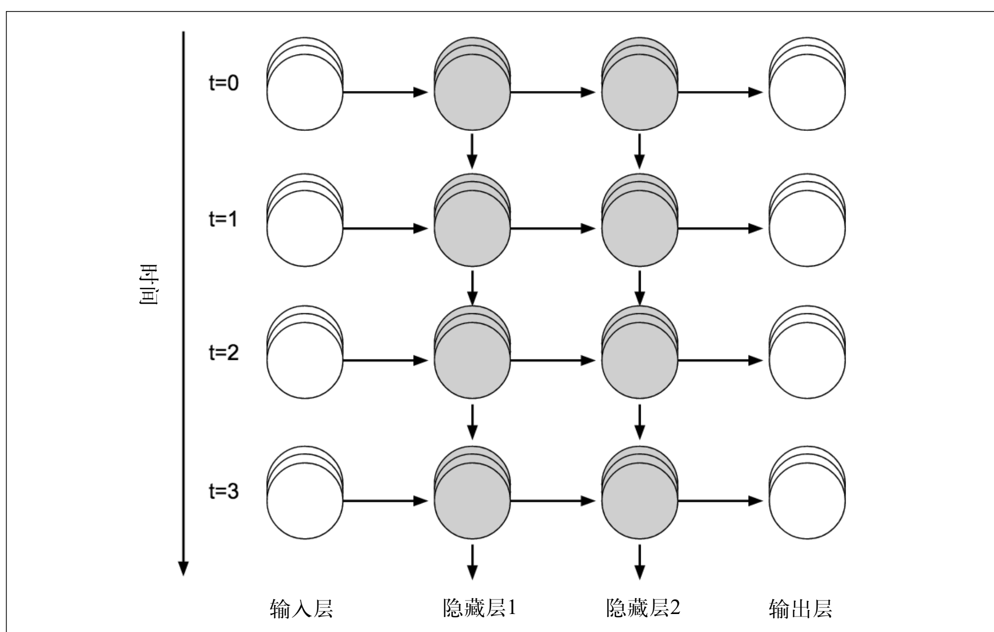


图 4-23：沿时间轴展开的递归神经网络

下面介绍，LSTM 网络通过循环连接比传统 RNN 传递更多信息。

3. LSTM单元

RNN 层的单元是传统人工神经元的变体。

每个 LSTM 单元都有两类连接。

- 与前一个时间步的连接（这些单元的输出）。
- 与前一层的连接。

LSTM 网络中的记忆单元是使网络能够随时间保持状态的核心概念。LSTM 单元的主体也称 **LSTM 块**，如图 4-24 所示。

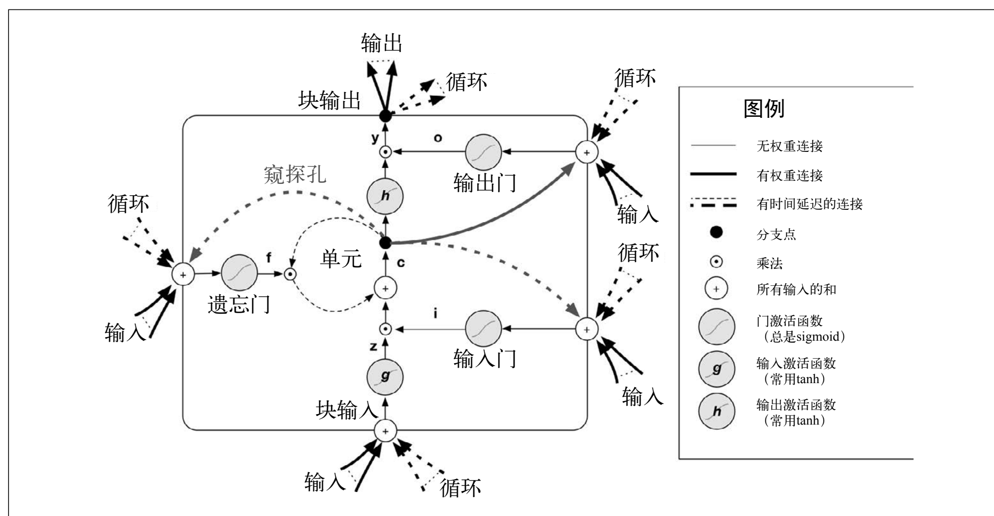


图 4-24：LSTM 块图

以下是 LSTM 单元的组件。

- 三种门
  - 输入门（输入调制门）
  - 遗忘门
  - 输出门
- 块输入
- 记忆单元（恒定误差转盘）
- 输出激活函数
- 窥探孔连接

以下三个门单元，用于保护线性单元免受误导信号影响。

- 输入门保护单元免受无关的输入事件影响。
- 遗忘门帮助单元忘记以前的记忆内容。
- 输出门在 LSTM 单元的输出里暴露（或不暴露）记忆单元的内容。

LSTM 块的输出与 LSTM 块的输入和所有门循环连接。LSTM 单元中输入门、遗忘门和输出门使用 sigmoid 激活函数进行 [0, 1] 限制。LSTM 块的输入和输出激活函数（通常）是一个 tanh 激活函数。



关于遗忘门的说明

1.0 的激活输出意味着“记住一切”，而 0.0 的激活输出意味着“遗忘一切”。从不同的角度看，遗忘门更好的名称可能是“记住门”！

考虑到这一点，通常将遗忘门的偏置初始化为大的值，以便学习长期依赖（在 DL4J 中默认的大值 ==1.0）。

图 4-25 列出了 LSTM 层前向传递的向量公式。图中使用了 Greff 等的表示方法。

$\boldsymbol{z}^t = g(\boldsymbol{W}_z\boldsymbol{x}^t + \boldsymbol{R}_z\boldsymbol{y}^{t-1} + \boldsymbol{b}_z)$	块输入
$\boldsymbol{i}^t = \sigma(\boldsymbol{W}_i\boldsymbol{x}^t + \boldsymbol{R}_i\boldsymbol{y}^{t-1} + \boldsymbol{p}_i \odot \boldsymbol{c}^{t-1} + \boldsymbol{b}_i)$	输入门
$\boldsymbol{f}^t = \sigma(\boldsymbol{W}_f\boldsymbol{x}^t + \boldsymbol{R}_f\boldsymbol{y}^{t-1} + \boldsymbol{p}_f \odot \boldsymbol{c}^{t-1} + \boldsymbol{b}_f)$	遗忘门
$\boldsymbol{c}^t = \boldsymbol{i}^t \odot z^t + \boldsymbol{f}^t \odot \boldsymbol{c}^{t-1}$	单元状态
$\boldsymbol{o}^t = \boldsymbol{\sigma}(\boldsymbol{W}_o\boldsymbol{x}^t + \boldsymbol{R}_o\boldsymbol{y}^{t-1} + \boldsymbol{p}_o \odot \boldsymbol{c}^t + \boldsymbol{b}_o)$	输出门
$\boldsymbol{y}^t = \boldsymbol{o}^t \odot h(\boldsymbol{c}^t)$	块输出

图 4-25：LSTM 层前向传递的向量公式

表 4-1 列出了图 4-25 中方程的各个变量。

表4-1：LSTM向量公式的变量说明

变量名	说　明
$\boldsymbol{x}^t$	时间 t 的输入向量
$\boldsymbol{W}$	矩形输入权重矩阵
$\boldsymbol{R}$	平方循环权重矩阵
$\boldsymbol{P}$	窥探孔权重向量
$\boldsymbol{b}$	偏置向量

自循环连接的固定权重（除了调制时）的值为 1.0，以克服梯度消失问题。这个核心单元使得 LSTM 单元能够发现序列中范围更大的事件。这些事件可以跨越最多 1000 个离散的时间步，相比之下，旧的循环架构只能对跨越 10 个时间步左右的事件建模。



关于更多的 LSTM 变体

请参阅论文“LSTM：A Search Space Odyssey”。

门控循环单元（GRU）

与 LSTM 类似的另一个循环单元是 GRU。GRU 具有重置门和更新门，类似于 LSTM 单元中的遗忘门 / 输入门。它们的主要区别在于，GRU 仅使用泄漏集成（但有由更新门控制的自适应时间常数）来完全暴露其记忆内容。GRU 受到了 LSTM 单元的启发，但它计算和实现起来更简单。

4. LSTM层

基本层接受输入向量 $\boldsymbol{x}$（非固定）并输出 y。输出 y 受输入 $\boldsymbol{x}$ 和所有输入历史的影响。该层通过循环连接，受到输入历史的影响。RNN 有一些内部状态，每当输入一个向量到该层时，状态就会被更新。状态由单个隐藏向量组成。

5. 训练

LSTM 网络使用监督学习更新网络中的权重。在一个向量序列中，它一次训练一个输入向量。向量为实值，是输入节点的激活值序列。每个非输入单元在任何给定的时间步计算其当前激活值。该激活值被计算为接收连接的所有单元激活值加权和的非线性函数。

对于输入序列中的每个输入向量，误差等于所有目标信号与网络计算相应激活值的偏差之和。下面介绍环神经网络的反向传播变体 BPTT，包括 LSTM 神经网络。

6. BPTT和截断BPTT

RNN 的训练计算成本高，传统上选择使用 BPTT。



BPTT 与标准反向传播基本相同，应用链式法则计算基于网络的连接结构的导数（梯度）。从某种意义上说，BPTT 是**穿越时间**的，其中一些梯度 / 误差信号也将从未来的时间步流回当前的时间步，而不仅仅是从上面的层（如标准反向传播中发生的）。

当循环网络处理有许多时间步的长序列时，推荐使用截断 BPTT。截断 BPTT 降低了 RNN 中更新每个参数的计算复杂度。

> **RNN 与反向传播**
>
> 在长度为 1000 的序列上计算 RNN 的梯度与在有 1000 层的多层感知器网络上进行前向和反向传递的计算成本相同。

执行更频繁的参数更新会加快 RNN 的训练。当输入序列超过几百个时间步时，建议使用截断 BPTT。

为了更好地理解截断 BPTT 所涉及的概念，设想当训练一个长度为 12（时间步）的时间序列输入的网络时会发生什么。在这种情况下，需要前向传递 12 步，然后计算网络的误差，之后，如图 4-26 所示，反向传递 12 个时间步。

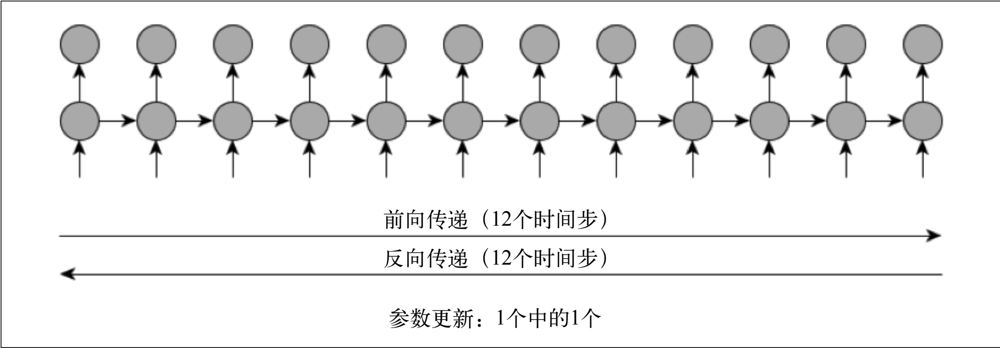


图 4-26：标准 BPTT

在图中，12 个时间步对于训练过程并不是那么困难。当模型的时间序列数据有几百个甚至更

多的时间步时，训练变得更加困难。如果时间序列输入的时间步数是 1000，那么标准反向传播训练对于每个前向传递和反向传递（对于每个单独的参数更新）都需要 1000 个时间步。计算成本很快就变高，这就是为什么考虑其他训练方法，如 BPTT 和截断 BPTT 的原因。

截断 BPTT 将前向和反向传递分割为较小的操作。图 4-27 所示的截断 BPTT 使用了较小的前向传递，以及同样小的反向传递，然后更新预期的参数。这个较小的传递大小是用户配置的超参数。在图中可以看到截断 BPTT 的传递大小是 4 个时间步。

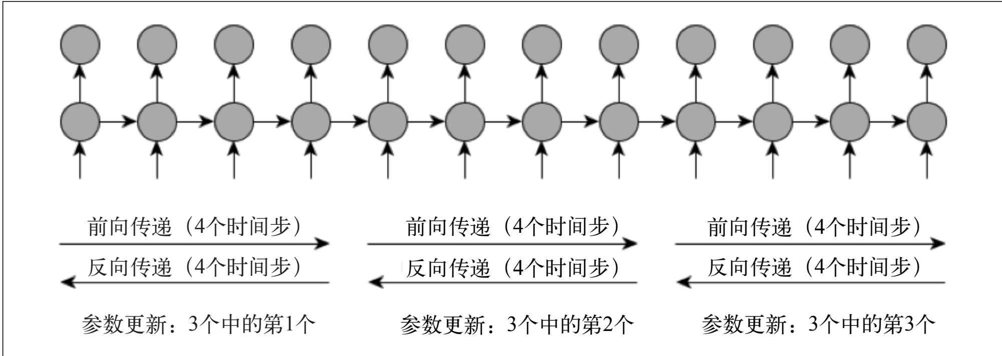


图 4-27：截断 BPTT

截断 BPTT 是目前训练 RNN 最实用的方法。与常规的 BPTT 相比，使用截断 BPTT，计算负担更小，却可以捕获更长的依赖。

标准 BPTT 和截断 BPTT 的总体复杂度相似，并且在训练期间需要相同数量的时间步。然而对于计算量相近的工作，却可以得到更多的参数更新（尽管每个参数更新都有一些开销）。与所有近似方法一样，使用截断 BPTT 时有轻微的缺点，截断 BPTT 中学到的依赖的长度最终可能比完整的 BPTT 算法的结果短。在实践中，只要截断 BPTT 长度被正确设置，这种速度上的取舍通常是值得的。

4.3.6 特定领域应用与混合网络

如前所述，RNN 执行许多特定领域的应用，如语音转录到文本、机器翻译和手写文本的生成。RNN 已被证明擅长计算机视觉领域，可以实现以下功能：

- 帧级视频分析；
- 为图像加字幕；
- 为视频加字幕；
- 视觉问题回答。

使用 RNN 研究计算机视觉的另一个新兴领域是一个通过每次只处理一小块区域来从图像中提取信息的网络，它被称为“视觉注意”的循环模型。这种模型能够高效地处理混有多个对象的图像，CNN 很难对这样的图像进行分类。这些网络与 CNN 结合使用，用于对原始感知器和 RNN 的时域建模。

另一个值得注意的 CNN+RNN 混合网络由 Andrej Karpathy 和李飞飞创建，能够生成图像

及由自然语言描述其区域的网络。这种模型使用图像及相应句子的数据集，能够生成图像的标题，如图 4-28 所示。

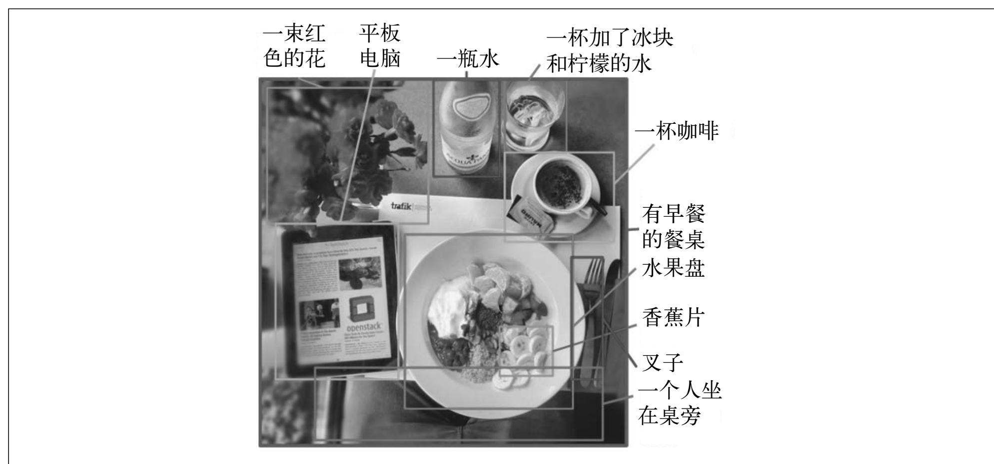


图 4-28：使用 CNN/RNN 混合模型标记图像

这类网络是 CNN 和 BRNN 的结合。

4.4 递归神经网络

递归神经网络类似于 RNN，可以处理可变长度的输入。二者的主要区别在于 RNN 能对训练数据集的层次结构建模。图像通常是一个由许多对象组成的场景。解构场景往往是一个令人感兴趣的问题领域，却不是一件简单的事情。这种解构的递归特性要求不仅要识别场景中的对象，还要识别对象与场景是如何关联的。

4.4.1 网络架构

递归神经网络结构由共享权重矩阵和二叉树结构组成，这个二叉树结构允许递归网络学习单词的不同序列或图像的不同部分。它是一个有用的句子和场景解析器。使用反向传递的递归神经网络的变体被称为**通过结构的反向传递**（BPTS）。前馈传递是自下而上，而反向传播是自上而下的。这里把目标视为树的顶端，而输入是底部。

4.4.2 递归神经网络的变体

递归神经网络有几种变体。一种是递归自动编码器。就像它的前馈表亲一样，递归自动编码器学习重建输入。在 NLP 的场景中，它学习重建上下文。半监督递归自动编码器在每个上下文学习某些标签的似然值。

另一种变体是被称为“递归神经张量网络”的监督神经网络，它在树的每个节点计算监督目标。“张量”意味着它计算梯度的方式稍有不同，利用张量（三维或更多维的矩阵）中信息的另一个维度，在每个节点容纳更多的信息。

4.4.3 递归神经网络的应用

递归和 RNN 共享许多相同的用例。RNN 传统上用于 NLP，因为它与二叉树、上下文和基于自然语言的解析器有关。例如，成分解析器能够将一个句子分解成二叉树，并根据该句子的语言属性对其分割。在使用递归神经网络时，强制使用构建树结构（通常是成分解析）的解析器。

递归神经网络可以恢复图像或句子等数据集中的粒度结构和高层次结构。其应用包括：

- 图像场景分解
- NLP
- 音频到文本转录

实践中看到的两个特定的网络配置是递归自动编码器和递归神经张量。使用递归自动编码器将句子分解成用于 NLP 的片段。使用递归神经张量将图像分解为它的组成对象，并对场景中的对象标注语义。

递归神经网络趋向于更快地训练，因此通常将它用在与时间更相关的应用中，但它已被证明在基于 NLP 的领域，如情感分析中也表现得很好。

4.5 小结与讨论

本章回顾了最新的深度学习的主要架构。这些架构总结和归类如下。

- 要生成数据（例如图像、音频或文本），我们将使用：
  - GAN
  - VAE
  - RNN
- 对图像建模，我们可以使用：
  - CNN
  - DBN
- 对序列数据建模，我们可以使用：
  - RNN/LSTM

接下来将展示这些网络的实际代码示例，以及探讨对不同类型神经网络训练和调优的方法。第 5 章将展示这些概念在 API 示例中是如何结合的，我们将在例子中实践 DL4J 深度学习库。在开始这些例子之前，首先讨论一些常见的有关深度学习的话题。

4.5.1 深度学习会使其他算法过时吗

围绕深度学习是否使其他建模算法过时的争论，在互联网的论坛中出现过很多次了。现在的答案是“否”，因为对于许多更简单的机器学习应用程序，那些更简单的算法对所需的模型准确度来说表现得很好，像逻辑回归这样的模型也更容易使用。因此在做决策时，需要根据领域所需的准确度来衡量投入的程度。然而，当对应用领域了解很少，并且难以进行高级手工特征构建时，深度学习算法往往表现良好。

4.5.2 不同的问题有不同的最佳方法

总的说来，机器学习是在适当的情况下应用正确的方法。目前还没有达到一种单一技术占主导地位的程度，因此每次都要评估问题的空间和数据，以找到最佳模型。这符合“没有免费午餐定理”。



没有免费午餐定理

“没有免费午餐定理”指出：没有一种模型适用于所有问题。对一个问题的一个伟大模型的假设可能不适用于另一个问题。在机器学习中，尝试多个模型并找到对某个特定的问题最有效的模型很常见。

每一种机器学习方法都有一定的偏差和方差。模型越接近真实的底层模型，我们就可以更好地用学习算法来完成。

理解这一点的另一个方式是从实际的例子来考虑它。如果数据明显是线性的，如可视化图形表示的那样，你会尝试用非线性模型（如多层感知器）来拟合数据吗？不，你可能以更简单的方法来处理这个问题，如逻辑回归。在 Kaggle 竞赛中，表现最好的方法因题目不同而不同。然而当深度学习不能获胜时，随机森林和集成学习方法往往是胜者。

输入数据集的大小可能是另一个影响深度学习是否适用于给定问题的因素。过去几年的实验结果表明，当数据集足够大时，深度学习的预测能力最好。这意味着随着数据集大小的增加，深度学习结果变得更好。神经网络比线性模型的表示能力更强，并且能够更好地利用数据。一个好的经验法则是，对于至少有 5000 个训练输入标签的例子，实践者应该训练神经网络。

4.5.3 什么时候需要深度学习

本章最后总结了一组简单的规则来帮助你回答这个问题：这个项目需要使用深度学习吗？

1. 何时使用深度学习

在以下场景，应该使用深度学习。

- 简单的模型（逻辑回归）无法达到用例所需的准确度。
- 在图像、NLP 或音频处理中有复杂的模式匹配。
- 有高维数据。
- 在向量中有时间维度（序列）。

2. 何时坚持传统的机器学习

在以下场景，应该使用传统的机器学习模型。

- 有高质量、低维度的数据，如从数据库导出的列式数据。
- 无须在图像数据中找到复杂的模式。

当数据不完整或质量差时，这两种方法得到的结果都很差。

第 5 章

建立深度网络

> 眼下可不是想你没有带什么东西的时候。想想你用手头现有的东西能做什么事儿吧。
>
> ——欧内斯特·海明威,《老人与海》

本章将介绍 DL4J 中可用的工具套件，以及一些可以在你自己的项目中使用的实例。首先讲解如何将特定的深度网络映射到适合的问题，之后深入研究库中的许多核心例子。



有关 DL4J 安装和技术支持的信息，请参阅附录 G。

5.1 将深度网络与适合的问题匹配

第 4 章介绍了深度学习的主题，以及跟对输入数据进行的手工特征工程相比，如何设计网络架构来匹配问题。本章将展示一些与特定类型的问题相匹配的深度网络的例子，基于这个目的，本章具体介绍以下应用：

- 列式数据建模
- 图像数据建模
- 序列 / 时间序列数据建模
- 自然语言处理应用

本章中的应用展示了从第 1 章就一直在构建的深度网络的概念。虽然没有第 4 章提到的架构的所有示例，但我们编写了一组示例演示深度学习的核心概念，这样你就可以轻松地将大多

数示例扩展到新的场景。首先看一下之前书中介绍的与网络架构正确匹配的数据类型。



在网上找到示例代码

我们在 GitHub 代码库中创建了示例代码的分支。即使 DL4J 这样的项目随着时间而发展，我们仍为你提供代码的快照，这些代码将始终与本书匹配。

5.1.1 列式数据与多层感知器

常见的列式数据具有静态的结构，它是 DL4J 中最适合经典的多层感知器神经网络的模型。这些问题可能受益于较小的特征工程，但通常网络可以自己找到数据集上的最佳权重。超参数调优是用多层感知器建模时的主要挑战之一。第 6 章将介绍多种技术来帮助和指导你选择超参数。

5.1.2 图像与CNN

CNN 已被证明擅长寻找原始图像数据的结构。历史上，图像建模领域主要由大量预处理技术统治，它们将输入图像规整并转换为建模技术能更好处理的形式。旋转或缩放的微小变化使图像处理成为一项艰巨的任务。CNN 使得网络能够处理原始的图像数据，让实践者集中可以精力调整网络架构。本章稍后将展示用 CNN 对手写数字进行分类的例子。



CNN 与原始图像数据

经过辛勤努力得到的列式数据，所有特征都蕴含在特定的列中。历史上，当特征不在模型所期望的正确的“点”时，机器学习建模将变得相对脆弱。

图像中的对象很少出现在我们希望它们出现的地方，这对经典机器学习技术的特征提取阶段而言是一个巨大的挑战。CNN 的一个主要优点是它们能够获取原始图像数据中处于任意形态场景中的对象，并擅长识别特征。

CNN 应用的发展

CNN 应用继续以惊人的速度发展。虽然推荐实践者从解决图像建模问题开始使用 CNN，但 CNN 网络也开始应用于文本建模问题，例如：

- 机器翻译
- 句子分类
- 情感分析

与在图像区域滑动窗口的方式相同，CNN 可以在字符序列上滑动窗口，这种特征定位不变性使得 CNN 成为深度网络工具箱中的有力工具。本章着重于一个对图像数据应用 CNN 的例子，不过需要注意的是，不应将其视作网络架构的限制，而应将其作为进一步研究的出发点。

5.1.3 时间序列与RNN

RNN 很适合处理序列化数据，如 Web 日志数据。如果序列化数据的每个样本都包含时间戳信息，这些数据就被视作时间序列数据。

时间序列和序列化数据绘制在二维图上时呈现为波形。通常我们寻找那些随着时间推移形成特定模式的图形区域，这与那些不具有时间域，而在二维网格中形成图片的图像数据形成对比。然而在时间序列和图像中，我们都在寻找数据中可能出现的特定对象。这些对象可能在大小上进行了缩放，并且通常不会每次都出现在数据中相同的位置，这就带来了挑战。

RNN 从多层感知器发展而来，用于更好地对时间序列数据的时间域建模。如同第 4 章中所看到的，RNN 允许将输入向量的序列当作 RNN 模型的单个逻辑输入，来更好地对时间域建模。



传感器、日志和其他测量值

在处理任何随时间测量的数据源时，使用 RNN 来更好地对数据在一段时间内的变化建模。

可以使用 RNN 进行分类、回归和生成新的输出。本章稍后将展示分类和新生成的例子。

扮演选择模型时的反对者

有时我们会听到这样的争论：一个实践者使用其他技术，如随机森林，取得了巨大成功。然而对于序列化数据，除非在特定的情况下，这通常不是好的做法。

一些实践者争辩说，他们不能像使用 RNN 那样，直接将随机森林应用于时间序列 / 序列数据，因为在对时间序列建模之前，需要把它转换成某种扁平的向量表示（例如特征提取）。

将深度学习模型与强大的模型 + 人工特征工程相比较的争论常常是一个陷阱。这是因为致力于特征工程的实践者可以持续地尝试发现更好的特征，直到发现一个特定的数据集中所有的特征（给予足够长的时间）。

深度学习对于时间序列（或任何实际问题）的真正优势基本如下所示。

如果采用定制的特征 / 算法的做法，我们需要为许多完全不同的算法 / 模型编写新代码，然后将其全部应用于每个新问题 / 数据集。如果处于纯软件工程领域，这种情况会是**技术性债务**。从历史上看，我们接受了这一机器学习的现状。

如果开始还不知道合适的特征，那么结合当前深度学习的最新技术，半自动化神经网络超参数调优会比半自动化特征工程更容易。一个精心设计的深度学习系统（包括超参数调优）可以高效地重复使用，并被期望在各种问题上表现良好。

当处理那些不确定但可能具有长期依赖的可变长度时间序列时，尤其适合使用 RNN。如果输入是固定长度的序列（即所有时间序列的长度都相同），那么也可以使用多层感知器或 CNN。

但如果你想输入任意长的序列，（假设）你需要考虑完整的输入数据历史，那么 RNN（或精心设计的具备大的隐藏状态空间的马尔可夫模型）基本上是唯一的选择。

5.1.4 使用混合网络

当遇到时间和图像相结合的数据（例如视频）时，应使用由 LSTM 和卷积层混合而成的特殊网络。我们推荐这种混合，因为 LSTM 网络能够捕获视频中随时间流逝、图像中变化的时间方面的信息，并且卷积层能够捕获帧数据本身的结构。

5.2 DL4J工具套件

DL4J 是一组深度学习工具的集合，它们被打包在一起作为一个套件来执行如下功能：

- 集成
- 向量化
- 建模
- 评估

这些工具被设计为可以在多个平台上工作，能以串行或并行方式执行。DL4J 从一开始就考虑到现代运行平台，并就此而设计，不会遇到其他机器学习库在过去十年中遇到的并行化问题。DL4J 社区还关注基本建模之外的需求，提供了与 Spark 和 Hadoop 分布式文件系统（HDFS）等平台更好的集成。DL4J 项目的向量化能力也很强大，引入 DataVec 作为该套件的一等公民。

DL4J 专注于企业级功能，目标用户是在深度学习中需要 JVM 选项，同时也需要 C++ 的速度和 Spark 并行计算能力的人。首先简单介绍 DL4J 套件的各个方面，然后介绍如何在你的笔记本上设置这些工具。

5.2.1 向量化与DataVec

由于神经网络只能训练向量，向量化数据是必要的预处理步骤。DL4J 套件包括一个名为 DataVec 的库，它允许我们快速创建适合核心 DL4J 库工具建模的向量化数据。本书稍后介绍 DataVec，它能够以本地模式工作，也可以在 Apache Spark 上以并行模式执行。

5.2.2 运行时与ND4J

ND4J 是 JVM 上的一个科学计算库。其语法模仿了 NumPy 和 MATLAB，还提供了可以使用 Java 进行线性代数和大规模矩阵操作的 *n* 维数组。ND4J 提供了一个整洁的数值接口，使用户无须改变实现代码，即可将后端处理从原生环境切换到 GPU。我们可以编写一次线

性代数的实现，然后在 Maven 中指定 ND4J 的后端。ND4J 最常见的后端有 x86 和 jcublas。DL4J 项目的开发最终将包括 OpenCL 和 Power 8。ND4J 包括 Java 和 Scala API，为基于 JVM 语言开发的程序员提供了熟悉的环境。ND4J 的 API 旨在提供功能，并且尽可能提供 NumPy 的简洁性和易用性。

DL4J Java API 允许程序员在可组合框架内配置自己的可扩展神经网络，以及根据需要调整超参数。它为神经网络的建立、调优、评估和数据加载提供了空间。ND4J 的 API 在类似于 NumPy 的环境中提供了基本的线性代数、微积分和信号处理功能（开发人员可以在多个 CPU 或 GPU 上部署分布式运行时）。Java API 和 ND4J 后端共同提供了当今企业级深度学习应用所需的编程环境和顶级速度。

1. ND4J与速度需求

在深度学习的世界里总是需要更快的计算。ND4J 提供了能够更快地执行线性代数运算的三个关键方式：

- 使用 JavaCPP
- CPU 后端
- GPU 后端

下面提供了一些说明，介绍这些关键领域的改进如何帮助 ND4J 加速深度模型训练。

- JavaCPP
  - 自动生成 C++ 的 JNI 绑定。
  - 允许在 Java 环境中轻松维护和部署 C++ 二进制文件。
- CPU 后端
  - OpenMP（本地操作中的多线程）。
  - OpenBLAS 或 MKL（BLAS 操作）。
  - SIMD 扩展。
- GPU 后端
  - DL4J 目前支持 Cuda 7.5（+cuBLAS），并且 Cuda 8.0 公布之后就会支持它。
  - 还可使用 cuDNN。



设置 GPU 支持

有关用 DL4J 设置 GPU 的详细信息，请参阅附录 I 中的指南。

2. ND4J和DL4J的性能基准测试

要了解 DL4J 与其他主要深度学习库比较的信息，请访问 GitHub 项目（https://github.com/deeplearning4j/dl4j-benchmark）。

表 5-1 是目前 DL4J 与主要库的比较结果。

表5-1：LeNet示例w/cuDNN

包	CPU	GPU	多线程	准确度
Dl4j	20m08s	3m13s	1m18s	~99.0%
Caffe	19m52s	53s	1m14s	~99.0%
Tensorflow	5m15s	1m44s	2m44s	~98.6%
Torch	18m03s	6m25s	3m50s	~98.3%

DL4J 社区继续推动 ND4J 和 DL4J 的性能不断向前发展。你可以随时查看公开的基准测试结果并重现它。

5.3 DL4J API的基本概念

本节将列出大多数 DL4J 示例使用的核心技术。如果想详细了解 API 的使用，请参考附录 E 或 DL4J 在线文档。

5.3.1 加载与保存模型

本节重点介绍一些你可能想用 DL4J API 做的事情。

1. 把训练好的模型写入磁盘

使用 `ModelSerializer` 类将模型的架构和参数写入磁盘。如以下代码片段所示：

```
BufferedOutputStream stream = new BufferedOutputStream(...);
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

这既适用于写入本地磁盘，也适用于写入其他文件系统，如 HDFS。

写入 HDFS。要写入 HDFS，只需使用正确的 `Path` 类（`org.apache.hadoop.fs.Path`）和格式正确的 HDFS 路径字符串，如下所示：

```
Path modelPath = new Path( hdfsPathToSaveModel );
BufferedOutputStream stream = new BufferedOutputStream( os );
ModelSerializer.writeModel( trainedNetwork, stream, true );
```

HDFS 文件目录通常如下所示：

```
hdfs:///path/to/my/file.txt
```

2. 从磁盘读取已保存的模型

如果需要从磁盘加载先前保存的 DL4J 模型，我们将再次使用 `ModelSerializer` 类，如下面的代码片段所示：

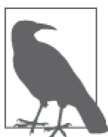
```
InputStream stream = ...
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

从 HDFS 读取。如果你想直接从 HDFS 加载模型，请使用相同的 API 调用，但是需要使用 Hadoop `FileSystem` 类创建 `InputStream`，如下所示：

```
org.apache.hadoop.conf.Configuration hadoopConfig = new org.apache.hadoop.conf
  .Configuration();
FileSystem hdfs = FileSystem.get(hadoopConfig);
InputStream stream = hdfs.open( new Path( hdfsPathToSavedModelFile ) );
MultiLayerNetwork network = ModelSerializer.restoreMultiLayerNetwork( stream );
```

5.3.2 为模型获取输入

DL4J 专门使用 NDArray 作为对模型进行训练和评分的数据结构。NDArray 与 `Dataset` 对象经常配合使用，用于模型的输入和输出。更多 `DataSet` 类用法的相关信息，请查看有关 ND4J 的附录。



数据向量化

如果想了解如何将原始数据转换为 DL4J 模型所需的 NDArray 向量，请查看附录 F 和附录 E。

训练中加载数据

DL4J 建模项目中，在开始训练数据的测试工作流之前，要先通过 `RecordReader` 从文件格式解析数据，然后将数据打包为小批量。

- `RecordReader`

  `RecordReader` 类的任务是从文件格式解析特定的向量，并将数据值转换为标准的 NDArray。

- `DataSetIterator`

  `DataSetIterator` 与 `RecordReader` 组合使用，接受每条记录产生的 NDArray，创建小批量的 NDArray 进行训练。



在 Spark 上加载数据

在 Spark 上加载数据的做法与之类似，但需要使用专门的类加载 Spark 和 HDFS 上的数据，第 9 章将详细讨论这些问题。

5.3.3 建立模型架构

用 DL4J 建模的每个神经网络都需要配置一个特定的架构。本书之前讨论了主要的架构，这些概念与你在 DL4J 中要设置的网络架构直接匹配。

1. 建立面向层的架构

`NeuralNetConfiguration` 对象是构建 DL4J 神经网络中层的基本对象。许多单个层的组合构成一个深度神经网络，如下面的例子所示：

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(1)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

```
        .learningRate(learningRate)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.RELU)
                .build())
        .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.SOFTMAX)
                .nIn(numHiddenNodes).nOut(numOutputs).build())
        .pretrain(false).backprop(true).build();
```

使用这个配置类，通过添加层并具体配置每一层，来设计预期的网络架构。

2. 超参数

`NeuralNetConfiguration` 对象还允许我们设置许多与训练相关的不同超参数，例如：

```
.learningRate(learningRate)
.updater(Updater.NESTEROVS).momentum(0.9)
```

在前面的代码片段中，可以看到超参数，如学习率、更新器和动量值是由给定的 `MultiLayerConfiguration` 对象设置的，这允许你控制网络如何被训练。本书稍后将探讨设置不同网络架构超参数的策略。

5.3.4 训练与评估

在配置了网络架构并通过 `RecordReader` 和 `Iterator` 加载数据之后，需要训练模型。我们会设置多轮，并在网络上调用 `.fit()` 方法，训练数据将作为参数传入，如下所示：

```
model.fit( trainingDataIter );
```

这个方法循环遍历由迭代器引用的数据集中所有的训练记录，并在网络完成循环之后返回。

每个训练轮被看作输入数据集的完全使用。在下面的代码片段中，可以看到一个在输入数据集迭代器上调用 `.fit()` 方法的模型。

```
for ( int n = 0; n < nEpochs; n++) {
    model.fit( trainingDataIter );
}
```

在本章稍后的例子中，我们会再次见到这种管理轮循环的模式。

1. 做出预测

为了更好地理解如何做出预测，建议阅读附录 E。

2. 训练数据、验证数据和测试数据

不仅将训练数据分成训练和测试部分，而且还包括验证部分，被视作一种最佳做法。使用验证部分的数据来指导我们尽可能早地结束训练。

5.4 使用多层感知器网络对CSV数据建模

对于新的深度学习实践者来说，刚开始学习 DL4J 可能感到很复杂，所以我们首先构建多层感知器模型，在你可能熟悉的旧神经网络架构环境中展示 DL4J API 的基本使用。本书附带的 GitHub 代码库中有一个 Java 实例，对一个被称为“Saturn”的合成非线性数据集建模。示例 5-1 是这个示例的代码（https://github.com/deeplearning4j/oreilly-book-dl4j-examples/blob/master/dl4j-examples/src/main/java/org/deeplearning4j/examples/feedforward/classification/MLPClassifierSaturn.java）。

示例 5-1　多层感知器示例

```
public class MLPClassifierSaturn {


    public static void main(String[] args) throws Exception {
        Nd4j.ENFORCE_NUMERICAL_STABILITY = true;
        int batchSize = 50;
        int seed = 123;
        double learningRate = 0.005;
        //轮数（数据完全传递）
        int nEpochs = 30;

        int numInputs = 2;
        int numOutputs = 2;
        int numHiddenNodes = 20;

        final String filenameTrain  =
            new ClassPathResource("/classification/saturn_data_train.csv")
                .getFile().getPath();
        final String filenameTest  =
            new ClassPathResource("/classification/saturn_data_eval.csv")
                .getFile().getPath();

        //加载训练数据：
        RecordReader rr = new CSVRecordReader();
        rr.initialize(new FileSplit(new File(filenameTrain)));
        DataSetIterator trainIter =
            new RecordReaderDataSetIterator(rr,batchSize,0,2);

        //加载测试/评估数据：
        RecordReader rrTest = new CSVRecordReader();
        rrTest.initialize(new FileSplit(new File(filenameTest)));
        DataSetIterator testIter =
            new RecordReaderDataSetIterator(rrTest,batchSize,0,2);

        //log.info("Build model....");
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
                .seed(seed)
                .iterations(1)
                .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
                .learningRate(learningRate)
                .updater(Updater.NESTEROVS).momentum(0.9)
                .list()
                .layer(0, new DenseLayer.Builder().nIn(numInputs)
                        .nOut(numHiddenNodes)
```

```
                    .weightInit(WeightInit.XAVIER)
                    .activation(Activation.RELU)
                    .build())
            .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                    .weightInit(WeightInit.XAVIER)
                    .activation(Activation.SOFTMAX)
                    .nIn(numHiddenNodes).nOut(numOutputs).build())
            .pretrain(false).backprop(true).build();


    MultiLayerNetwork model = new MultiLayerNetwork(conf);
    model.init();
model.setListeners(new ScoreIterationListener(10));    //每更新10个参数就
                                                       //打印分数

    for ( int n = 0; n < nEpochs; n++) {
        model.fit( trainIter );
    }

    System.out.println("Evaluate model....");
    Evaluation eval = new Evaluation(numOutputs);
    while(testIter.hasNext()){
        DataSet t = testIter.next();
        INDArray features = t.getFeatureMatrix();
        INDArray lables = t.getLabels();
        INDArray predicted = model.output(features,false);

        eval.eval(lables, predicted);

    }


    System.out.println(eval.stats());
    //---------------------------------------------------------------------------
    //训练已结束。接下来的代码只是用于画出数据的图形以及预测


    double xMin = -15;
    double xMax = 15;
    double yMin = -15;
    double yMax = 15;

    //在x/y输入空间的每个点评估预测值，并做好绘制图的准备工作
    int nPointsPerAxis = 100;
    double[][] evalPoints = new double[nPointsPerAxis*nPointsPerAxis][2];
    int count = 0;
    for( int i=0; i<nPointsPerAxis; i++ ){
        for( int j=0; j<nPointsPerAxis; j++ ){
            double x = i * (xMax-xMin)/(nPointsPerAxis-1) + xMin;
            double y = j * (yMax-yMin)/(nPointsPerAxis-1) + yMin;

            evalPoints[count][0] = x;
            evalPoints[count][1] = y;

            count++;
        }
    }
```

```
        INDArray allXYPoints = Nd4j.create(evalPoints);
        INDArray predictionsAtXYPoints = model.output(allXYPoints);

        //将所有的训练数据放在一个数组里，并绘制它：
        rr.initialize(new FileSplit(new File(filenameTrain)));
        rr.reset();
        int nTrainPoints = 500;
        trainIter = new RecordReaderDataSetIterator(rr,nTrainPoints,0,2);
        DataSet ds = trainIter.next();
        PlotUtil.plotTrainingData(ds.getFeatures(), ds.getLabels(), allXYPoints,
            predictionsAtXYPoints, nPointsPerAxis);


        //获取测试数据，通过网络运行测试数据，并进行预测，然后绘制这些预测：
        rrTest.initialize(new FileSplit(new File(filenameTest)));
        rrTest.reset();
        int nTestPoints = 100;
        testIter = new RecordReaderDataSetIterator(rrTest,nTestPoints,0,2);
        ds = testIter.next();
        INDArray testPredicted = model.output(ds.getFeatures());
        PlotUtil.plotTestData(ds.getFeatures(), ds.getLabels(), testPredicted,
            allXYPoints, predictionsAtXYPoints, nPointsPerAxis);

        System.out.println("****************Example finished********************");
    }

}
```

下面解释代码的不同部分是如何一起工作，来对 Saturn 数据集建模的。

5.4.1 建立输入数据

输入数据是非线性数据集，其形式如下所示：

```
1,-7.1239700674365,-5.05175898010314
0,1.80771566423302,0.770505522143023
1,8.43184823707231,-4.2287794074931
0,0.451276074541732,0.669574142606103
0,1.52519959303934,-0.953055551414968
```

第一列是行的标签，第二列和第三列表示两个独立变量列。我们需要使用 DL4J 迭代器引用数据集，因此使用 `CSVRecordReader` 来加载数据。

```
//加载训练数据：
RecordReader rr = new CSVRecordReader();
rr.initialize(new FileSplit(new File(filenameTrain)));
DataSetIterator trainIter = new RecordReaderDataSetIterator(rr,batchSize,0,2);
```

在迭代器中，我们告诉记录读取器数据有几列、哪些列表示标签。

5.4.2 确定网络架构

我们想要一个基本的多层感知器，并且需要使用 `MultiLayerConfiguration` 对象来设置它（以及任何 DL4J 网络架构），如下所示：

```
 //log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(1)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(learningRate)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.RELU)
                .build())
        .layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
                .weightInit(WeightInit.XAVIER)
                .activation(Activation.SOFTMAX)
                .nIn(numHiddenNodes).nOut(numOutputs).build())
        .pretrain(false).backprop(true).build();


MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
model.setListeners(new ScoreIterationListener(10)); //每更新10个参数就打印分数
```

这个例子中有两层：

- `DenseLayer`
- `OutputLayer`

下面了解网络架构中特定的部分。

1. 通用超参数

我们使用以下参数设置优化算法：

```
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
```

这指示 DL4J：我们打算使用 SGD 优化算法。学习率设置如下：

```
.learningRate(learningRate)
```

这指示 DL4J：我们打算使用动量因子为 0.9 的 Nesterov 参数更新策略：

```
.updater(Updater.NESTEROVS).momentum(0.9)
```

2. 第一个隐藏层

第一个隐藏层从向量化管道产生的输入中获取原始值。这些值经过各种形式的规范化之后，通常在 [−1.0, 1.0] 或 [0.0, 1.0] 范围内。

```
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.RELU)
        .build())
```

这里层的输入神经元需要与输入向量中自变量列的数量相同。

```
.nIn(numInputs)
```

这一层的输出数就是神经网络下一层神经元的数量。在这个例子中，这个数由变量 numHiddenNodes 表示。对于该层，我们还使用 WeightInit.XAVIER 策略来初始化权重，激活函数是修正线性函数。

```
.weightInit(WeightInit.XAVIER)
.activation(Activation.RELU)
```

下面研究这个网络的下一层——输出层。

3. 用于分类的输出层

这个输出层使用的是一个 softmax 输出激活函数，回顾一下之前介绍多标签 softmax 输出层使用的章节。在这个问题中，我们正在构建一个二元分类器，因此也可以使用 sigmoid 输出激活函数。

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
```

预期的输入数量与先前（输入）层的输出数量相同。输出单元的数量是两个，因为这是一个二元分类器，而 softmax 会给出每个类别的分数。



sigmoid 还是 softmax 输出层

在数学上使用 sigmoid 单输出与使用有两个输出单元的 softmax 输出层（使用 DL4J 的 MCXENT/NegativeLogLikelihood，以及独热表示法：写法是 [1, 0] 或 [0, 1]，而不是 [0] 或 1）相同。下面的专栏介绍了独热向量表示法。

这里还指定输出层使用负对数似然损失函数，因为它通常与 softmax 输出层配合使用。

独热向量表示

独热表示法的做法是：给定一组位，该向量中只有一列的值可以为 1.0，其他所有列的值都为 0.0。这种向量表示方法经常被用来表示使用“1 of *K*”（*K* 个类别之一）分类方案的整数特征。表 5-2 给出了值（0 到 4）的常规二进制表示以及独热表示的示例。

表5-2：独热向量的形象化表示

值	二进制编码	独热编码
0	000	00000001
1	001	00000010
2	010	00000100
3	011	00001000
4	100	00010000

5.4.3 训练模型

为了训练模型，我们建立一个 for 循环，它将在输入数据集上以指定轮（如遍历整个数据集）数训练神经网络。

```
for ( int n = 0; n < nEpochs; n++) {
    model.fit( trainIter );
}
```

为了在整个数据集上训练，我们调用 MultiLayerNetwork 类实例的 .fit() 方法。这个类处理被适当指定的超参数的基本机制，比如之前在输入数据集迭代器中配置的小批量的大小。

```
DataSetIterator testIter = new RecordReaderDataSetIterator(rrTest,batchSize,0,2);
```

在这段代码中，batchSize 变量控制一个批量中从磁盘收集多少样本传递到模型中训练。当模型被训练时，控制台中出现类似于下面这样的输出：

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 0.6313823699951172
o.d.o.l.ScoreIterationListener - Score at iteration 10 is 0.4763660430908203
o.d.o.l.ScoreIterationListener - Score at iteration 20 is 0.42963680267333987
o.d.o.l.ScoreIterationListener - Score at iteration 30 is 0.39850467681884766
o.d.o.l.ScoreIterationListener - Score at iteration 40 is 0.3672478103637695
```

误差数会随着时间减少，当它接近 0.0 时，就说明模型几乎完成了对训练数据的学习。

5.4.4 评估模型

下面的代码示例用于评估新多层感知器模型。这里通过 testIter 对象实例，将实际标签和预测标签加载到 Evaluation 类的实例中。

```
System.out.println("Evaluate model....");
Evaluation eval = new Evaluation(numOutputs);
while(testIter.hasNext()){
    DataSet t = testIter.next();
    INDArray features = t.getFeatureMatrix();
    INDArray lables = t.getLabels();
    INDArray predicted = model.output(features,false);

    eval.eval(lables, predicted);

}

System.out.println(eval.stats());
```

再回顾一下第 1 章 F1 评分和其他与评估相关的指标的概念。eval.stats() 方法调用将产生以下控制台输出：

```
Evaluate model....

Examples labeled as 0 classified by model as 0: 48 times
Examples labeled as 1 classified by model as 1: 52 times
```

```
==========================Scores========================================
 Accuracy:  1
 Precision: 1
 Recall:    1
 F1 Score:  1
========================================================================
```

这是一个相对简单的数据集，在 30 轮之后，DL4J 能够在所有评估项上得到完美的评分（1.0）。下面转向更复杂的例子。

5.5 利用CNN对手写图像建模

正如本章开头所探讨的那样，CNN 擅长图像分类。下个例子将展示如何对手写的数字图像进行加载和建模，建模得到的 CNN 模型能够对没见过的新的手写数字图像进行分类。

在这个例子中，训练数据集和测试数据集设置有数据集迭代器。我们将在训练数据集上训练数据，然后用留出的测试数据集评估模型的准确度。我们使用的训练数据集是 **MNIST 手写图像数据集**。

稍后你将看到这个模型的架构与前面的示例不同，它具有不同的层和参数。这个特殊的 CNN 被称为 LeNet。



LeNet

LeNet 卷积架构聚焦于一系列的卷积层及随后的最大池化层。（示例 5-2 展示了这个 CNN 架构在 DL4J 中的实现）

5.5.1 使用LeNet CNN的Java代码示例

示例 5-2 提供了 MNIST LeNet CNN 示例的代码（http://bit.ly/2toXqtn）。

示例 5-2 DL4J 中为 MNIST 构建的 LeNet

```
public class LenetMnistExample {
    private static final Logger log = LoggerFactory
        .getLogger(LenetMnistExample.class);

    public static void main(String[] args) throws Exception {
        int nChannels = 1; //输入的通道数
        int outputNum = 10; //可能的输出数
        int batchSize = 64; //测试批量的大小
        int nEpochs = 1; //训练轮数
        int iterations = 1; //训练迭代数
        int seed = 123; //

        /*
            为要迭代的数据集生成批量大小的迭代器
         */
        log.info("Load data....");
        DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
        DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);
```

```
/*
    构建神经网络
 */
log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations) //如上训练迭代
        .regularization(true).l2(0.0005)
        /*
            为学习率衰减和偏置打开注释
         */
        .learningRate(.01)//.biasLearningRate(0.02)
        //.learningRateDecayPolicy(LearningRatePolicy.Inverse)
        //.lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
        .weightInit(WeightInit.XAVIER)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new ConvolutionLayer.Builder(5, 5)
                //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用
                //的过滤器的数量
                .nIn(nChannels)
                .stride(1, 1)
                .nOut(20)
                .activation(Activation.IDENTITY)
                .build())
        .layer(1, new SubsamplingLayer
            .Builder(SubsamplingLayer.PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(2, new ConvolutionLayer.Builder(5, 5)
                //注意nIn无须应用到后面的层
                .stride(1, 1)
                .nOut(50)
                .activation(Activation.IDENTITY)
                .build())
        .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer
            .PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
        .layer(5, new OutputLayer
             .Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .build())
        .setInputType(InputType.convolutionalFlat(28,28,1)) //查看下面的说明
        .backprop(true).pretrain(false).build();

/*
.setInputType(InputType.convolutionalFlat(28,28,1)) 这一行做了以下几件事情。
```

```
    (a) 它增加了预处理器，用于处理卷积/子采样层和密集层之间的转换等事情。
    (b) 做一些额外的配置验证。
    (c) 必要时，根据前一层的大小（但不会覆盖用户手动设置的值）设置每一层的nIn
        (输入神经元的数量，或使用CNN时的输入深度）值。
        InputTypes也可用于其他类型的层（RNN、MLP等），而不仅仅是CNN。
    对于普通的图像，（当使用ImageRecordReader时）使用InputType.convolutional
(height,width,depth)。
    MNIST记录读取器是一种特殊情况，它以"扁平化"行向量格式（即1x784向量）输出28x28像
素的灰度（nChannels=1）图像，因此这里使用的输入类型是"convolutionalFlat"。
        */

        MultiLayerNetwork model = new MultiLayerNetwork(conf);
        model.init();


        log.info("Train model....");
        model.setListeners(new ScoreIterationListener(1));
        for( int i=0; i<nEpochs; i++ ) {
            model.fit(mnistTrain);
            log.info("*** Completed epoch {} ***", i);

            log.info("Evaluate model....");
            Evaluation eval = new Evaluation(outputNum);
            while(mnistTest.hasNext()){
                DataSet ds = mnistTest.next();
                INDArray output = model.output(ds.getFeatureMatrix(), false);
                eval.eval(ds.getLabels(), output);

            }
            log.info(eval.stats());
            mnistTest.reset();
        }
        log.info("****************Example finished********************");
    }
}
```

下面介绍此程序特定的部分，以及它们是如何协同工作来对 MNIST 图像数据集建模的。

5.5.2 加载及向量化输入图像

这个例子使用了一个名为 `MnistDataSetIterator` 的自定义数据集迭代器。这是因为 MNIST 数据集是自定义的二进制格式，这些文件不是通常所期望的那样：目录中包含一组单独的 JPG 或 PNG 文件。为了简化这个示例，你只需知道程序在幕后执行必要的函数，原始图像数据被提取到 NDArray 中以便使用 DL4J 训练。以下的代码片段展示了如何创建 `MnistDataSetIterator`。

```
DataSetIterator mnistTrain = new MnistDataSetIterator(batchSize,true,12345);
DataSetIterator mnistTest = new MnistDataSetIterator(batchSize,false,12345);
```

在这个代码片段中，代码使用单独的迭代器分别加载训练和测试数据集。这个程序还将自动从互联网上下载 MNIST 数据集，并在本地解压缩以供使用。

5.5.3 DL4J中用于LeNet的网络架构

我们再次看到使用 `MultiLayerConfiguration` 对象描述前面多层感知器示例中的网络架构。不过请注意，下面的代码示例所示的网络具有更多的层，并且层的类型不同于多层感知器的示例。

```
/*
    构建神经网络
 */
log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations) //如上训练迭代
        .regularization(true).l2(0.0005)
        /*
            为学习率衰减和偏置打开注释
         */
        .learningRate(.01)//.biasLearningRate(0.02)
        //.learningRateDecayPolicy(LearningRatePolicy.Inverse)
        .lrPolicyDecayRate(0.001).lrPolicyPower(0.75)
        .weightInit(WeightInit.XAVIER)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .list()
        .layer(0, new ConvolutionLayer.Builder(5, 5)
                //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的
                //过滤器的数量
                .nIn(nChannels)
                .stride(1, 1)
                .nOut(20)
                .activation(Activation.IDENTITY)
                .build())
        .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(2, new ConvolutionLayer.Builder(5, 5)
                //注意nIn无须应用到后面的层
                .stride(1, 1)
                .nOut(50)
                .activation(Activation.IDENTITY)
                .build())
        .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                .kernelSize(2,2)
                .stride(2,2)
                .build())
        .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
        .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .build())
        .setInputType(InputType.convolutionalFlat(28,28,1)) //查看下面的说明
```

```
        .backprop(true).pretrain(false).build();

/*
.setInputType(InputType.convolutionalFlat(28,28,1)) 这一行做了几件事情。
(a) 它增加了预处理器，用于处理卷积/子采样层和密集层之间的转换之类的事情。
(b) 做一些额外的配置验证。
(c) 必要时，根据前一层的大小（但不会覆盖用户手动设置的值）设置每一层的nIn
    （输入神经元的数量，或使用CNN时的输入深度）值。
InputTypes也可以用于其他类型的层（RNN、MLP等），而不仅仅是CNN。
对于普通的图像（当使用ImageRecordReader时）使用InputType.convolutional
(height,width,depth)。
MNIST记录读取器是一种特殊情况，它以"扁平化"行向量格式（即1x784向量）输出
28x28像素的灰度（nChannels=1）图像，因此这里使用的输入类型是
"convolutionalFlat"。
*/

MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

1. 通用超参数

下面的代码片段中一些主要的超参数：

```
.seed(seed)
.iterations(iterations) //如上训练迭代
.regularization(true).l2(0.0005)
/*
    为学习率衰减和偏置打开注释
*/
.learningRate(.01)//.biasLearningRate(0.02
//.learningRateDecayPolicy(LearningRatePolicy.Inverse).lrPolicyDecayRate(0.001).
//lrPolicyPower(0.75)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
```

下面的列表解释了如何设置这些超参数以及我们为它们设置的值。

- 正则化

  在这里，正则化被开启（设置为 true），L2 正则化的参数被设置为 0.0005。

- 权重初始化

  对于这个 LeNet 架构的示例，Xavier 权重初始化策略表现良好。

- 优化算法

  优化算法被设置为 SGD。在很多深度学习的例子中，SGD 很常见，因为它在很多情况下都表现良好。第 6 章和第 7 章将讨论优化算法的变体。

- 更新器

  我们选择 Nesterov 作为这个例子的更新器，其中 Nesterov 被大量使用，因为它在实践中效果很好。当这些更新都在一个方向上进行时，通常 Nesterov 会增加更新的规模。可以想象一下，这种影响就像我们要登一座平缓但非常矮的山，方向是明确的，但步子可能需要更大些。

2. 卷积层

在本节开始的代码示例中我们可以看到卷积层的一般模式和第 4 章描述的最大池化层。在下面的示例中，我们会看到 LeNet 网络中的第一个卷积层。

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的过滤器的数量
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
```

在这个示例中，使用了 Java 建造者模式设置卷积层的属性，来了解其中每个属性。

- 过滤器的大小

  在这个示例中，我们为这个层创建一个 5×5 大小的过滤器。

- 输入数据通道

  这个示例的通道数量被设置为 1，因为自定义数据集迭代器将原始图像自动转换为黑白图像。其他示例也许需要定义表示图像数据中 RGB 通道的三个输入通道。

- 步长

  该层的步长被设置为 `(1,1)`，这意味着当过滤器在输入空间滑动时，每次只需要向右，然后向下移动一步。

- 激活函数

  我们使用恒等函数作为该卷积层的输出。



注意恒等激活函数

LeNet 网络架构（1998）比 ReLU（2012）出现得早。卷积层改用 ReLU 后加速了 SGD 的收敛。出于历史原因，这个例子保持了原来 LeNet 架构的完整性。



卷积层与激活函数

在最近的 CNN 中，经常使用 ReLU 激活函数建立卷积层。

3. 最大池化层

下面是紧跟在第一个卷积层之后的池化层的代码。

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

以下为层属性。

- 最大池化层

  该层被设置为最大池化层，这个池化层有一个大小为 (2,2) 的过滤器，这意味着从前一层的 5×5 过滤器下采样到当前层的 2×2 网格。

- 设置池化层步长

  过滤器的步长被设置为 (2,2)，这意味着它将在水平滑动时前进两个值，然后向下移动到下面的行时前进两个值。

4. 输出层

因为正在构建（例如数字 0~9）的分类模型具有两个以上标签，所以我们需要一个使用 softmax 激活函数的输出层，如下所示：

```
.layer(5, new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLIKELIHOOD)
        .nOut(outputNum)
        .activation(Activation.SOFTMAX)
        .build())
```

该输出层使用负对数似然损失函数（使用 softmax 输出层时常见），并且输出单元的数量等于该数据集的类别或标签的数量。

5.5.4 训练CNN网络

现在已经建立了 LeNet 卷积模型架构，我们可以初始化 `MultiLayerNetwork` 对象以在输入数据集上进行训练。下面的代码片段展示如何从上一节中获取配置并设置网络。

```
MultiLayerNetwork model = new MultiLayerNetwork(conf);
model.init();
```

通过配置模型，现在可以在输入 MNIST 数据集上进行所需轮数的训练，如下所示：

```
log.info("Train model....");
model.setListeners(new ScoreIterationListener(1));
for( int i=0; i<nEpochs; i++ ) {
    model.fit(mnistTrain);
    log.info("*** Completed epoch {} ***", i);

    log.info("Evaluate model....");
    Evaluation eval = new Evaluation(outputNum);
    while(mnistTest.hasNext()){
        DataSet ds = mnistTest.next();
        INDArray output = model.output(ds.getFeatureMatrix(), false);
        eval.eval(ds.getLabels(), output);

    }
    log.info(eval.stats());
    mnistTest.reset();
}
log.info("****************Example finished********************");
```

在训练的循环中，使用了与 MNIST 数据集迭代器将小批量图像传给模型的 `.fit()` 方法的相同的模式。在 `.fit()` 方法完成当前轮之后，我们根据测试 MNIST 数据集检查模型的训

练情况。这样就能看到随着训练一轮一轮的进行，F1 分数逐步变化。随着时间的推移，应该会看到评估分数上升，损失函数误差下降。

5.6 基于RNN的序列数据建模

本节通过两个例子展示 RNN 的生成和判别能力：生成莎士比亚风格的作品和分类时间序列。

5.6.1 通过LSTM生成莎士比亚风格作品

从一个有趣的例子——对莎士比亚的作品建模开始。在这个例子中，我们训练一个 LSTM RNN 生成莎士比亚的作品（准确地说是“莎士比亚风格的作品”）。注意，我们将文本视为字符序列，并且模型基于它过去遇到的字符预测下一个最有可能的字符。你还可以调整该模型以处理其他常见的序列数据，比如将在下一个示例中看到的日志数据或传感器数据。



RNN 不可思议的有效性

这个例子部分受到了 Andrej Karpathy 的博文“The Unreasonable Effectiveness of Recurrent Neural Networks”的启发。



训练 LSTM 的源文本

这个例子基于从古腾堡计划[1]下载的《威廉·莎士比亚全集》进行训练，基于其他文本来源的训练应该也比较容易实现。

1. 高级别的建模工作流

这个例子介绍在 DL4J 中建立 RNN 架构的概念，其建立在我们从之前的例子中学到的一些概念的基础上，比如：

- 加载输入数据集训练；
- 配置网络架构。

我们还将介绍专门支持 RNN 的 DL4J API 的一些新的组件。这个例子学习莎士比亚作品的每个字符，然后，我们要求模型基于学到的模式生成新的原创作品。

运行这个例子时，可以看到输出不断变化，一开始产生的输出如下所示：

```
----- Sample 0 -----
lnee!
  Lhir tape shepyang? Nocw; mame. Budt hlant'nthely ler ild
   Py theu sfochill'ad my and ocs im nereepapd werer;
  Motadid. Mert hatterhirl. Iit    nesdoesd'nlowhednanieivetranns deugheuind
   Bred yetide rathane fojlond thivh uweet.
   Thy lametom theuegfast lart souclalitoloe ilntangylrt or
```

注 1：威廉·莎士比亚全集（https://www.gutenberg.org/ebooks/100），文件大小 5.3MB，UTF-8 编码，约 540 万个字符。

```
----- Sample 1 -----
l,, ne agly
   Lot Bolncanbom bavantenfircasle womlidibl.
  NTERIOO. IrdmisfUoItolleeeddortiss hot buye.
   The hetenle of ile,
   'merlliydingiponI, bomgule? Shurtstarer of ate,
   Onbibly ot ire pomxatgillant, dakl.
   Oxt Mtanlonfye wiudsimotime raugadent deu'y ondtstes.
   If vonee.
 Whol touEde
```

之后产生的输出如下所示：

```
----- Sample 0 -----
ous reward me, Master Warce! I-will stay
    shall; for I one as mine lord.
  CLOTEN. Come, I will thigh, i'; and what wam! Hath dravelly
    The albowed out, Aside dismernicges could be a
    druck than there's thoughts, here is we with me and rag.
    Thou shalt love it doth my child.
  PERDITA. Ti

----- Sample 1 -----
on,
    Incie Paties, go, thurst with thy flounds by the bands. Exit COURSTIO
  FLORIZEL. Uncle, an if you,
    Abassom the man,
    Stars, you spite-hath loved.
  QUEEN MANGER On stay is! Who is mer?
  CLOTEN. Hang't, what I'll remain,
    Cap nothes same so here;
    My tens
```

下面来看看这个例子的 Java 代码。

2. 对莎士比亚作品建模的Java代码

示例 5-3 列出了示例的完整代码（http://bit.ly/2sUs2PU）。

示例 5-3　利用 DL4J 的 LSTM 对莎士比亚作品建模及生成

```
public class GravesLSTMCharModellingExample {
    public static void main( String[] args ) throws Exception {
        int lstmLayerSize = 200;                      //每个GravesLSTM层中的单元数
        int miniBatchSize = 32;                       //训练时用到的小批量的大小
        int exampleLength = 1000;                     //要用的每个样本序列的大小
                                                      //这个值当然可以增大
        int tbpttLength = 50;                         //截断基于时间的反向传播长度，
                                                      //即每50个字符做一次参数更新
        int numEpochs = 1;                            //总的训练轮数
        int generateSamplesEveryNMinibatches = 10;    //从网络生成样本的频度。1000字
                                                      //符/50tbptt长度：每个小批量更
                                                      //新20个参数
        int nSamplesToGenerate = 4;                   //每轮训练后生成的样本的数量
        int nCharactersToSample = 300;                //生成的每个样本的长度
        String generationInitialization = null;       //可选的字符初始化，如果为空，
                                                      //那么会使用随机字符
```

```
//上面的代码被用来"事先指导"LSTM使用一个字符序列去继续/结束。
//初始化字符默认必须全部在CharacterIterator.getMinimalCharacterSet()中。
Random rng = new Random(12345);

//得到一个DataSetIterator，它将文本向量存入那些可以用于训练GravesLSTM网络
//的对象中。
//GravesLSTM网络
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
int nOut = iter.totalOutcomes();

//设置网络配置：
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns())
         .nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
            .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX)   //用于分类的MCXENT+softmax
            .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

MultiLayerNetwork net = new MultiLayerNetwork(conf);
net.init();
net.setListeners(new ScoreIterationListener(1));

//（为每层）打印网络中参数的数量
Layer[] layers = net.getLayers();
int totalNumParams = 0;
for( int i=0; i<layers.length; i++ ){
    int nParams = layers[i].numParams();
    System.out
        .println("Number of parameters in layer " + i + ": " + nParams);
    totalNumParams += nParams;
}
System.out.println("Total number of network parameters: " + totalNumParams);

//进行训练，然后生成和打印来自网络的样本
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
```

```
            net.fit(ds);
            if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
                System.out.println("--------------------");
                System.out.println("Completed " + miniBatchNumber +
                    " minibatches of size " + miniBatchSize + "x" + exampleLength
                        + " characters" );
                System.out.println("Sampling characters from network
                    given initialization \"" +
               (generationInitialization == null ? "" :
               generationInitialization) + "\"");
                 String[] samples = sampleCharactersFromNetwork(
                     generationInitialization,net,iter,rng,nCharactersToSample,
                         nSamplesToGenerate);
                 for( int j=0; j<samples.length; j++ ){
                     System.out.println("----- Sample " + j + " -----");
                     System.out.println(samples[j]);
                     System.out.println();
                 }
             }
         }

         iter.reset();    //为另一轮重置迭代器
     }

    System.out.println("\n\nExample complete");
}

/** 下载莎士比亚训练数据，并保存在本地（临时文件夹）。
 * 然后设置和返回一个简单的、基于文本进行向量化的DataSetIterator。
 * @param miniBatchSize 每个训练小批量的文本段的数量
 * @param sequenceLength 每个文本段中的字符数。
 */
public static CharacterIterator getShakespeareIterator(int miniBatchSize,
    int sequenceLength) throws Exception{
    //威廉·莎士比亚全集
    //文件大小5.3MB，UTF-8编码，约540万个字符。
    //https://www.gutenberg.org/ebooks/100
    String url = "https://s3.amazonaws.com/dl4j-distribution/pg100.txt";
    String tempDir = System.getProperty("java.io.tmpdir");
    String fileLocation = tempDir + "/Shakespeare.txt";    //下载文件的存储位置
    File f = new File(fileLocation);
    if( !f.exists() ){
        FileUtils.copyURLToFile(new URL(url), f);
        System.out.println("File downloaded to " + f.getAbsolutePath());
    } else {
        System.out.println("Using existing text file at " + f.getAbsolutePath());
    }

    if(!f.exists()) throw new IOException("File does not exist: " +
        fileLocation);    //下载问题？

    //允许哪些字符？其他的将被移除
    char[] validCharacters = CharacterIterator.getMinimalCharacterSet();
    return new CharacterIterator(fileLocation, Charset.forName("UTF-8"),
            miniBatchSize, sequenceLength, validCharacters, new Random(12345));
```

```
}

/** 根据给定初始化值（可选，有可能是null），从网络生成样本。
 * 初始化值被用来"事先指导"RNN使用一个你想拓展/继续处理的序列。
 * 注意初始化值用于所有样本。
 * @param initialization 字符串，可能为null。如果为null，则为所有样本选择一个
 * 随机字符作为初始化值
 * @param charactersToSample 要从网络中采样的字符数（排除初始化值）
 * @param net 带有一个或多个GravesLSTM/RNN层的MultiLayerNetwork，以及一个
 * softmax输出层
 * @param iter CharacterIterator. 用于从索引回到字符
 */
private static String[] sampleCharactersFromNetwork(String initialization,
                                                    MultiLayerNetwork net,
                                                    CharacterIterator iter,
                                                    Random rng,
                                                    int charactersToSample,
                                                    int numSamples ){
    //设置初始化值。如果没有设置初始化值，则使用一个随机字符
    if( initialization == null ){
        initialization = String.valueOf(iter.getRandomCharacter());
    }

    //初始化输入
    INDArray initializationInput = Nd4j.zeros(numSamples, iter.inputColumns(),
        initialization.length());
    char[] init = initialization.toCharArray();
    for( int i=0; i<init.length; i++ ){
        int idx = iter.convertCharacterToIndex(init[i]);
        for( int j=0; j<numSamples; j++ ){
            initializationInput.putScalar(new int[]{j,idx,i}, 1.0f);
        }
    }

    StringBuilder[] sb = new StringBuilder[numSamples];
    for( int i=0; i<numSamples; i++ ) sb[i] = new StringBuilder(initialization);

    //从网络采样（并将样本反馈到输入），一次一个字符（对所有样本）
    //这里的采样并行执行
    net.rnnClearPreviousState();
    INDArray output = net.rnnTimeStep(initializationInput);
    output = output.tensorAlongDimension(output.size(2)-1,1,0); //取得上一个
                                                                //时间步的输出

    for( int i=0; i<charactersToSample; i++ ){
        //通过对前一个输出采样，来设置下一个输入（一个时间步）
        INDArray nextInput = Nd4j.zeros(numSamples,iter.inputColumns());
        //输出呈概率分布。对于每个我们想生成的样本，从输出采样，然后加到新的输
        //入中
        for( int s=0; s<numSamples; s++ ){
            double[] outputProbDistribution = new double[iter.totalOutcomes()];
            for( int j=0; j<outputProbDistribution.length; j++ )
                outputProbDistribution[j] = output.getDouble(s,j);
            int sampledCharacterIdx =
                sampleFromDistribution(outputProbDistribution,rng);
```

```
                nextInput.putScalar(new int[]{s,sampledCharacterIdx}, 1.0f);
                //准备下一个时间步的输入
                sb[s].append(iter.convertIndexToCharacter(sampledCharacterIdx));
                //把采样的字符加到StringBuilder（人类可读的输出）
            }
            output = net.rnnTimeStep(nextInput);    //执行一次前向传递的时间步
        }

        String[] out = new String[numSamples];
        for( int i=0; i<numSamples; i++ ) out[i] = sb[i].toString();
        return out;
    }

    /** 给定离散类别上的概率分布，从分布中采样并返回生成的类别的索引
     * @param distribution 离散类别上的概率分布。总和必须为1.0
     */
    public static int sampleFromDistribution( double[] distribution, Random rng ){
        double d = rng.nextDouble();
        double sum = 0.0;
        for( int i=0; i<distribution.length; i++ ){
            sum += distribution[i];
            if( d <= sum ) return i;
        }
        //如果分布是有效的概率分布，这里不会被执行
        throw new IllegalArgumentException("Distribution is invalid? d="+d+",
                sum="+sum);
    }
}
```

下面会讨论这段代码，并解释其主要部分如何协同工作来对莎士比亚作品进行建模和生成。

3. 输入数据的建立与向量化

输入数据由示例中包含的支持类自动下载并转换为 NDArray，如下面的代码片段所示：

```
CharacterIterator iter = getShakespeareIterator(miniBatchSize,exampleLength);
```

对其有兴趣的读者可研究它如何在幕后工作。

4. LSTM网络架构

就像前两个例子一样，这里使用相同的构造器模式建立 LSTM 网络的层。

```
//配置网络：
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
```

```
.layer(0, new GravesLSTM.Builder().nIn(iter.inputColumns())
    .nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
.layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX)    //用于分类的MCXENT+softmax
        .nIn(lstmLayerSize).nOut(nOut).build())
.backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
.pretrain(false).backprop(true)
.build();
```

关于超参数的通用说明。在这个例子中，SGD 再次用作优化算法，学习率设置为 `0.1`，正则化是打开的，L2 设置为 `0.001`，更新器是 `RMSPROP`，这一点与迄今为止我们看到的其他例子不同。

- 隐藏层

  对输出层之外的层使用特殊的 GravesLSTM 层和 anh 激活函数。

- 输出层

  输出层与我们目前为止所看到的不同，因为使用了特殊的 `RnnOutputLayer` 处理 RNN 可能给出的不同类型的输出，如下所示：

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX)        //用于分类的MCXENT+softmax
         .nIn(lstmLayerSize).nOut(nOut).build())
```

即使再次在输出层使用了 softmax 激活函数，这里使用的损失函数依然是 `LossFunction.MCXENT`。



RMSProp 在实践中的应用

RMSProp 的设计对更新规模具有规范化作用，这意味着使用此更新器时，量级不同的参数（不同层，或同一层中的不同参数）在更新后，最终大小将大致相同。

5. 训练LSTM网络

下面的代码示例演示了如何训练网络。

```
//进行训练，然后生成和打印来自网络的样本
int miniBatchNumber = 0;
for( int i=0; i<numEpochs; i++ ){
    while(iter.hasNext()){
        DataSet ds = iter.next();
        net.fit(ds);
        if(++miniBatchNumber % generateSamplesEveryNMinibatches == 0){
            System.out.println("--------------------");
            System.out.println("Completed " + miniBatchNumber +
                " minibatches of size " + miniBatchSize +
                "x" + exampleLength + " characters" );
            System.out.println("Sampling characters from network given
```

```
                initialization \"" + (generationInitialization == null ? "" :
                generationInitialization) + "\"");
            String[] samples =
                sampleCharactersFromNetwork(
                    generationInitialization,net,iter,rng,nCharactersToSample,
                    nSamplesToGenerate);
            for( int j=0; j<samples.length; j++ ){
                System.out.println("----- Sample " + j + " -----");
                System.out.println(samples[j]);
                System.out.println();
            }
        }
    }

    iter.reset();    //为另一轮重置迭代器
}

System.out.println("\n\nExample complete");
```

这个示例使用 DL4J API 的方式有点不同。与迭代器本身调用 `.fit()` 方法相反，这里它在小批量上被显式调用，使得我们可以更好地控制在小批量上调用 `.fit()` 之间所能做的事情。这段代码让我们不用考虑莎士比亚数据集的输入过程，而从网络生成样本。

随着训练的进行，从控制台的输出可以看出，训练期间损失函数误差逐渐减少。

```
o.d.o.l.ScoreIterationListener - Score at iteration 0 is 217.28348109866505
o.d.o.l.ScoreIterationListener - Score at iteration 1 is 213.24020789706773
o.d.o.l.ScoreIterationListener - Score at iteration 2 is 212.96001041971766
o.d.o.l.ScoreIterationListener - Score at iteration 3 is 175.06079409241767
o.d.o.l.ScoreIterationListener - Score at iteration 4 is 165.25272077487378
```

这个示例中出现了与前面的代码片段类似的输出，正在进行的网络周期性地输出生成的句子样本。

6. 生成莎士比亚作品的样本

为了从网络生成样本，我们调用示例中列出的支持方法，该方法从模型生成一段合成的文本，如下所示：

```
String[] samples = sampleCharactersFromNetwork(generationInitialization,net,iter,
    rng,nCharactersToSample,nSamplesToGenerate);
for( int j=0; j<samples.length; j++ ){
    System.out.println("----- Sample " + j + " -----");
    System.out.println(samples[j]);
    System.out.println();
}
```

这是在训练期间周期性向控制台打印输出的代码的一部分。

5.6.2　基于LSTM的传感器时间序列分类

下面看一个使用 LSTM RNN 对序列进行分类的例子。我们使用加州大学尔湾分校（UCI）机器学习库的合成控制图表时间序列数据集。这个代码示例将建立一个模型来将单变量时

间序列分为以下六类：

- 正常（C）
- 循环（B）
- 增长趋势（E）
- 下降趋势（A）
- 上移（D）
- 下移（F）

图 5-1 展示了以上各类的示例图像渲染。

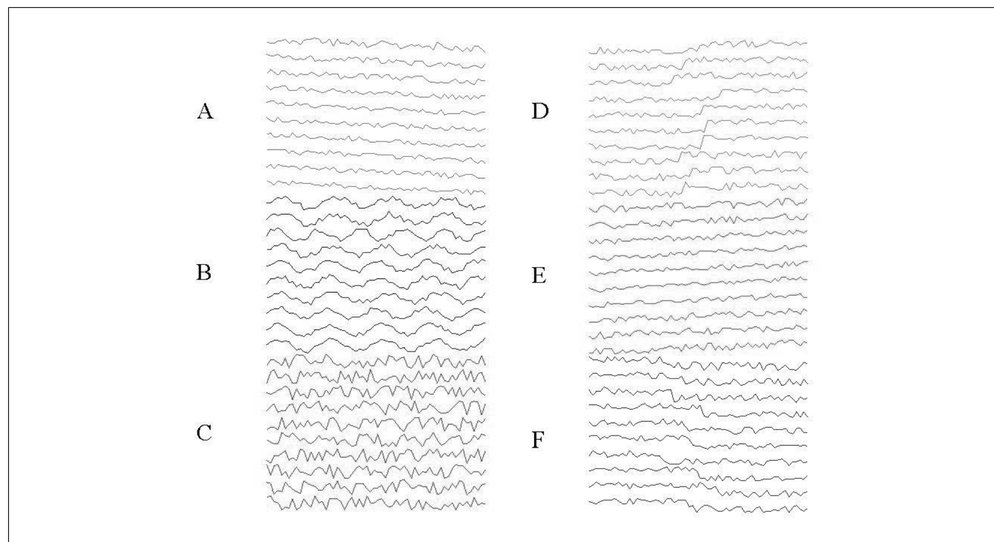


图 5-1：来自 UCI 库的合成时间序列示例

这六类序列是在嵌入式传感器领域生成的真实数据的一个很好的例子，给了我们一个实用的数据集来操作。

1. 循环分类示例的Java代码列表

示例 5-4 给出了下一个示例的 Java 代码（http://bit.ly/2tZM946），即循环分类示例，代码中建立了一个模型来对“UCI 合成控制图表时间序列数据集”中的序列进行分类。

示例 5-4　UCI 序列分类示例的 Java 代码

```
public class UCISequenceClassificationExample {
    private static final Logger log = LoggerFactory
        .getLogger(UCISequenceClassificationExample.class);

    //'baseDir'：数据保存的父目录。如果想把数据保存到其他目录，则修改它的值。
    private static File baseDir = new File("src/main/resources/uci/");
    private static File baseTrainDir = new File(baseDir, "train");
    private static File featuresDirTrain = new File(baseTrainDir, "features");
    private static File labelsDirTrain = new File(baseTrainDir, "labels");
    private static File baseTestDir = new File(baseDir, "test");
```

```
private static File featuresDirTest = new File(baseTestDir, "features");
private static File labelsDirTest = new File(baseTestDir, "labels");

public static void main(String[] args) throws Exception {
    downloadUCIData();

    //----- 加载训练数据 -----
    //注意我们有450个要训练的特征文件：从train/features/0.csv到train/
    //features/449.csv
    SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
    trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
        .getAbsolutePath() + "/%d.csv", 0, 449));
    SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
    trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
        .getAbsolutePath() + "/%d.csv", 0, 449));

    int miniBatchSize = 10;
    int numLabelClasses = 6;
    DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(
        trainFeatures, trainLabels, miniBatchSize, numLabelClasses,
        false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

    //规范化训练数据
    DataNormalization normalizer = new NormalizerStandardize();
    normalizer.fit(trainData);              //收集训练数据统计信息
    trainData.reset();

    //使用前面收集的统计信息来进行动态规范化。由'trainData'迭代器返回的每个
    //数据集都会被规范化。
    trainData.setPreProcessor(normalizer);


    //----- 加载测试数据 -----
    //与训练数据相同的处理过程。
    SequenceRecordReader testFeatures = new CSVSequenceRecordReader();
    testFeatures.initialize(new NumberedFileInputSplit(featuresDirTest
         .getAbsolutePath() + "/%d.csv", 0, 149));
    SequenceRecordReader testLabels = new CSVSequenceRecordReader();
    testLabels.initialize(new NumberedFileInputSplit(
        labelsDirTest.getAbsolutePath() + "/%d.csv", 0, 149));

    DataSetIterator testData = new SequenceRecordReaderDataSetIterator(
        testFeatures, testLabels, miniBatchSize, numLabelClasses,
        false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

    testData.setPreProcessor(normalizer);   //注意这里使用的正是对训练数据使用
                                            //的同一个规范化过程


    //----- 配置网络 -----
    MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(123)    //随机数字生成的种子，用于提升可重复性。可选。
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
                .iterations(1)
            .weightInit(WeightInit.XAVIER)
```

```
            .updater(Updater.NESTEROVS).momentum(0.9)
            .learningRate(0.005)
            .gradientNormalization(GradientNormalization
                .ClipElementWiseAbsoluteValue)  //并非每次都需要，不过对这个数
                                                //据集有用
            .gradientNormalizationThreshold(0.5)
            .list()
            .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
                .nOut(10).build())
            .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction
                .MCXENT)
                    .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
                        .build())
            .pretrain(false).backprop(true).build();

    MultiLayerNetwork net = new MultiLayerNetwork(conf);
    net.init();

    net.setListeners(new ScoreIterationListener(20)); //每执行20次迭代打印
                                                      //一次分数（损失函数值）


    //----- 训练网络，每轮训练后都评估测试数据的表现 -----
    int nEpochs = 40;
    String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
    for (int i = 0; i < nEpochs; i++) {
        net.fit(trainData);

        //使用测试数据集评估：
        Evaluation evaluation = net.evaluate(testData);
        log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

        testData.reset();
        trainData.reset();
    }

    log.info("----- Example Complete -----");
}


//这个方法下载数据，并将其从"每行一个时间序列"的格式转化为
//DataVec(CsvSequenceRecordReader)和DL4J可读的合适的CSV序列形式。
private static void downloadUCIData() throws Exception {
    if (baseDir.exists()) return;    //如果数据已存在，不要再次下载

    String url =
        "https://archive.ics.uci.edu/ml/machine-learning-databases/
         synthetic_control-mld/synthetic_control.data";
    String data = IOUtils.toString(new URL(url));

    String[] lines = data.split("\n");

    //创建目录
    baseDir.mkdir();
    baseTrainDir.mkdir();
```

```
        featuresDirTrain.mkdir();
        labelsDirTrain.mkdir();
        baseTestDir.mkdir();
        featuresDirTest.mkdir();
        labelsDirTest.mkdir();

        int lineCount = 0;
        List<Pair<String, Integer>> contentAndLabels = new ArrayList<>();
        for (String line : lines) {
            String transposed = line.replaceAll(" +", "\n");

            //标签：前100个样本（行）的标签是0，第二组100个样本的标签是1，以此类推
            contentAndLabels.add(new Pair<>(transposed, lineCount++ / 100));
        }

        //随机分配训练和测试数据：
        Collections.shuffle(contentAndLabels, new Random(12345));

        int nTrain = 450;   //75%训练，25%测试
        int trainCount = 0;
        int testCount = 0;
        for (Pair<String, Integer> p : contentAndLabels) {
            //在适当的路径以可读的形式写入输出
            File outPathFeatures;
            File outPathLabels;
            if (trainCount < nTrain) {
                outPathFeatures = new File(featuresDirTrain, trainCount + ".csv");
                outPathLabels = new File(labelsDirTrain, trainCount + ".csv");
                trainCount++;
            } else {
                outPathFeatures = new File(featuresDirTest, testCount + ".csv");
                outPathLabels = new File(labelsDirTest, testCount + ".csv");
                testCount++;
            }

            FileUtils.writeStringToFile(outPathFeatures, p.getFirst());
            FileUtils.writeStringToFile(outPathLabels, p.getSecond().toString());
        }
    }
}
```

下面分别解释代码的各个部分。

2. 设置输入数据和向量化

在示例 5-4 中，不需要手动获取数据，因为 `downloadUCIData()` 方法会帮我们做这件事。它还将 600 个时间序列样本分成含 450 个样本的训练集和含 150 个样本的测试集。之后，为了进行序列分类，它使用 `CSVSequenceRecordReader` 加载数据，然后将数据写为适合加载的格式。这种格式的每个文件都有一个时间序列，标签在另一个单独的文件中。

例如 train/features/0.csv 表示特征的记录为 0，而文件 train/labels/0.csv 表示标签的记录为 0。因为数据是单变量时间序列，所以在 CSV 文件中只有一列。通常每列包含多个值（例如每行一个时间步）。此外，因为每个时间序列只有一个标签，所以标签 CSV 文件只包含单个值。

如果只看如下所示的加载训练数据部分的代码，可以看到如何使用DL4J中的 SequenceRecordReader 读取 CSV 文件[2]，创建标签，然后创建一个 DataSetIterator 来处理这种特定类型的序列数据。

```
//----- 加载训练数据 -----
//注意我们有450个要训练的特征文件：从train/features/0.csv到train/features/449.csv
SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));
SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;
DataSetIterator trainData = new SequenceRecordReaderDataSetIterator(trainFeatures,
    trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

//规范化训练数据
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData);              //收集训练数据统计信息
trainData.reset();

//使用前面收集的统计信息来进行动态规范化。由'trainData'迭代器返回的每个数据集都
//会被规范化。
trainData.setPreProcessor(normalizer);
```

在这段代码最后，可以看到 DataNormalizer 对象被创建来收集数据集统计数据。这使得我们能够收集全局数据集统计数据，并规范化训练数据，以便更好地训练。

3. 网络架构与训练

我们的神经网络的基础架构是 LSTM RNN，在下面的代码片段中它与 MultiLayerConfiguration 对象一起被设置。

```
//----- 配置网络 -----
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(123)    //随机数字生成的种子，用于提升可重复性。可选。
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.NESTEROVS).momentum(0.9)
        .learningRate(0.005)
        .gradientNormalization(GradientNormalization
            .ClipElementWiseAbsoluteValue)  //并非每次都需要，不过对这个数据集有用
        .gradientNormalizationThreshold(0.5)
        .list()
        .layer(0, new GravesLSTM.Builder().activation(Activation.TANH).nIn(1)
            .nOut(10).build())
        .layer(1, new RnnOutputLayer.Builder(LossFunctions.LossFunction.MCXENT)
                .activation(Activation.SOFTMAX).nIn(10).nOut(numLabelClasses)
                    .build())
        .pretrain(false).backprop(true).build();
```

注 2： 有关这一步的更多细节，请访问 http://deeplearning4j.org/usingrnns#data。

这里只需要一个连接到 softmax 输出层的单个 LSTM 层，使用 `XAVIER` 权重初始化方法与 `NESTEROVS` 更新策略，并设置学习率为 `0.005`。

如同其他代码示例一样，执行多轮训练，直到误差率下降到一定程度，如以下代码所示：

```
//----- 训练网络，每轮训练后都评估测试数据的表现 -----
int nEpochs = 40;
String str = "Test set evaluation at epoch %d: Accuracy = %.2f, F1 = %.2f";
for (int i = 0; i < nEpochs; i++) {
    net.fit(trainData);

    //使用测试数据集评估：
    Evaluation evaluation = net.evaluate(testData);
    log.info(String.format(str, i, evaluation.accuracy(), evaluation.f1()));

    testData.reset();
    trainData.reset();
}

log.info("----- Example Complete -----");
```

这里再次调用了 `.fit()` 方法训练向量化和规范化后的输入训练集。之后使用 `Evaluation` 对象来衡量神经网络模型能否较好地泛化留出的测试数据。

5.7　利用自动编码器检测异常

为了演示自动编码器的实际使用，我们将展示使用简单的没有预训练的自动编码器，来对 MNIST 数据集执行异常检测。

检测目标是识别离群数字，即那些不寻常或不像典型数字的数字。本例使用重建误差来实现这一目标：常规样本的重建误差应该较低，而离群值的重建误差应该较高。

这个例子中配置模型的方式与其他例子相同，但它们的模型架构不同。在自动编码器的大多数层使用 `DenseLayer`（全连接），但是我们主要关注那些向下“过滤”，然后“扩展”恢复到输出层的层。

5.7.1　自动编码器示例的Java代码列表

示例 5-5 是执行异常检测的自动编码器的 Java 代码示例（http://bit.ly/2tQiQAP）。

示例 5-5　用于查找异常值的自动编码器的 Java 代码

```
public class MNISTAnomalyExample {

    public static void main(String[] args) throws Exception {

        //设置网络。784 输入/输出（由于MNIST图像是28x28）
        //784 -> 250 -> 10 -> 250 -> 784
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
                .seed(12345)
                .iterations(1)
                .weightInit(WeightInit.XAVIER)
```

```
            .updater(Updater.ADAGRAD)
            .activation(Activation.RELU)
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .learningRate(0.05)
            .regularization(true).l2(0.0001)
            .list()
            .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
                    .build())
            .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
                    .build())
            .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
                    .build())
            .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
                    .lossFunction(LossFunctions.LossFunction.MSE)
                    .build())
            .pretrain(false).backprop(true)
            .build();

    MultiLayerNetwork net = new MultiLayerNetwork(conf);
    net.setListeners(Collections.singletonList((IterationListener) new
        ScoreIterationListener(1)));

    //加载数据，并将其分割为训练集和测试集。40 000个用作训练，10 000个用作测试
    DataSetIterator iter = new MnistDataSetIterator(100,50 000,false);

    List<INDArray> featuresTrain = new ArrayList<>();
    List<INDArray> featuresTest = new ArrayList<>();
    List<INDArray> labelsTest = new ArrayList<>();

    Random r = new Random(12345);
    while(iter.hasNext()){
        DataSet ds = iter.next();
        SplitTestAndTrain split = ds.splitTestAndTrain(80, r);  //80/20分割
                                                                //（设一个小
                                                                //批量=100）
        featuresTrain.add(split.getTrain().getFeatureMatrix());
        DataSet dsTest = split.getTest();
        featuresTest.add(dsTest.getFeatureMatrix());
        INDArray indexes = Nd4j.argMax(dsTest.getLabels(),1);  //从独热编码表
                                                               //示转换为索引
        labelsTest.add(indexes);
    }

    //训练模型：
    int nEpochs = 30;
    for( int epoch=0; epoch<nEpochs; epoch++ ){
        for(INDArray data : featuresTrain){
            net.fit(data,data);
        }
        System.out.println("Epoch " + epoch + " complete");
    }

    //基于测试数据评估模型
    //分别对测试集中每个数字/样本评分
    //然后将三个数据的集合Triple（分数、数字和INDArray数据）添加到列表中并按
```

```
    //分数排序
    //这使我们可以得到每类最好的N个和最差的N个数字
    Map<Integer,List<Triple<Double,Integer,INDArray>>> listsByDigit =
        new HashMap<>();
    for( int i=0; i<10; i++ ) listsByDigit.put(i,new ArrayList<Triple<Double,
        Integer,INDArray>>());

    int count = 0;
    for( int i=0; i<featuresTest.size(); i++ ){
        INDArray testData = featuresTest.get(i);
        INDArray labels = labelsTest.get(i);
        int nRows = testData.rows();
        for( int j=0; j<nRows; j++){
            INDArray example = testData.getRow(j);
            int label = (int)labels.getDouble(j);
            double score = net.score(new DataSet(example,example));
            listsByDigit.get(label).add(new ImmutableTriple<>(score, count++,
                example));
        }
    }

    //分别对于每种数字，根据分数对数据排序
    Comparator<Triple<Double, Integer, INDArray>> c
        = new Comparator<Triple<Double, Integer, INDArray>>() {
        @Override
        public int compare(Triple<Double, Integer, INDArray> o1, Triple<Double,
            Integer, INDArray> o2) {
            return Double.compare(o1.getLeft(),o2.getLeft());
        }
    };

    for(List<Triple<Double, Integer, INDArray>> list : listsByDigit.values()){
        Collections.sort(list, c);
    }

    //为每种数字选择5个最好的和5个最差的数字（通过重建误差）
    List<INDArray> best = new ArrayList<>(50);
    List<INDArray> worst = new ArrayList<>(50);
    for( int i=0; i<10; i++ ){
        List<Triple<Double,Integer,INDArray>> list = listsByDigit.get(i);
        for( int j=0; j<5; j++ ){
            best.add(list.get(j).getRight());
            worst.add(list.get(list.size()-j-1).getRight());
        }
    }

    //最好和最差数字的可视化表示
    MNISTVisualizer bestVisualizer = new MNISTVisualizer(2.0,best,"Best
        (Low Rec. Error)");
    bestVisualizer.visualize();

    MNISTVisualizer worstVisualizer = new MNISTVisualizer(2.0,worst,"Worst
        (High Rec. Error)");
    worstVisualizer.visualize();
}

public static class MNISTVisualizer {
```

```
        private double imageScale;
        private List<INDArray> digits;  //数字（以行向量的形式），一个INDArray一个数字
        private String title;
        private int gridWidth;

        public MNISTVisualizer(double imageScale, List<INDArray> digits,
            String title ) {
            this(imageScale, digits, title, 5);
        }

        public MNISTVisualizer(double imageScale, List<INDArray> digits,
            String title, int gridWidth ) {
            this.imageScale = imageScale;
            this.digits = digits;
            this.title = title;
            this.gridWidth = gridWidth;
        }

        public void visualize(){
            JFrame frame = new JFrame();
            frame.setTitle(title);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            JPanel panel = new JPanel();
            panel.setLayout(new GridLayout(0,gridWidth));

            List<JLabel> list = getComponents();
            for(JLabel image : list){
               panel.add(image);
            }

            frame.add(panel);
            frame.setVisible(true);
            frame.pack();
        }

        private List<JLabel> getComponents(){
            List<JLabel> images = new ArrayList<>();
            for( INDArray arr : digits ){
                BufferedImage bi = new BufferedImage(28,28,BufferedImage
                    .TYPE_BYTE_GRAY);
                for( int i=0; i<784; i++ ){
                    bi.getRaster().setSample(i % 28, i / 28, 0, (int)(255*arr
                        .getDouble(i)));
                }
                ImageIcon orig = new ImageIcon(bi);
                Image imageScaled = orig.getImage().getScaledInstance((int)
                    (imageScale*28),(int)(imageScale*28),Image.SCALE_REPLICATE);
                ImageIcon scaled = new ImageIcon(imageScaled);
                images.add(new JLabel(scaled));
            }
            return images;
        }
    }
}
```

下面分别解释代码的各个部分。

5.7.2 设置输入数据

下面是使用自定义迭代器加载 MNIST 数据集部分的示例代码。

```
//加载数据，并将其分割为训练集和测试集。40 000个用于训练，10 000个用于测试
DataSetIterator iter = new MnistDataSetIterator(100,50 000,false);

List<INDArray> featuresTrain = new ArrayList<>();
List<INDArray> featuresTest = new ArrayList<>();
List<INDArray> labelsTest = new ArrayList<>();

Random r = new Random(12345);
while(iter.hasNext()){
    DataSet ds = iter.next();
    SplitTestAndTrain split = ds.splitTestAndTrain(80, r); //80/20分割
                                                           //（设一个小批量=100）
    featuresTrain.add(split.getTrain().getFeatureMatrix());
    DataSet dsTest = split.getTest();
    featuresTest.add(dsTest.getFeatureMatrix());
    INDArray indexes = Nd4j.argMax(dsTest.getLabels(),1); //从独热编码表示转换为索引
    labelsTest.add(indexes);
}
```

与前面的示例相比，这个示例以不同的方式处理训练和测试数据，演示了如何以不同的形式使用 DL4J 和 ND4J API。在 `while` 循环中，我们手动将数据集分成测试数据集和训练数据集。

5.7.3 自动编码器的网络结构与训练

如前所述，自动编码器网络架构的形状通常像一个漏斗，之后在输出层再扩展回完整的输入数据集大小。我们使用自动编码器学习输入数据最有效的形式。自动编码器如其名所示，它学着最好地代表数据。设置自动编码器的代码如下所示：

```
//设置网络。784 输入/输出（由于MNIST图像是28x28）
//784 -> 250 -> 10 -> 250 -> 784
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(12345)
        .iterations(1)
        .weightInit(WeightInit.XAVIER)
        .updater(Updater.ADAGRAD)
        .activation(Activation.RELU)
        .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .learningRate(0.05)
        .regularization(true).l2(0.0001)
        .list()
        .layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
                .build())
        .layer(1, new DenseLayer.Builder().nIn(250).nOut(10)
                .build())
        .layer(2, new DenseLayer.Builder().nIn(10).nOut(250)
                .build())
        .layer(3, new OutputLayer.Builder().nIn(250).nOut(784)
                .lossFunction(LossFunctions.LossFunction.MSE)
```

```
            .build())
        .pretrain(false).backprop(true)
        .build();
```

这个架构有四层，最后一层具有输入层中输入数据的全部 784 个单元。我们把这个网络中所有激活函数都设置为 ReLU，因为它最适合处理这个数据集。

训练自动编码器网络时，使用与本章其他 DL4J 示例一样的通用模式，如下所示：

```
//训练模型：
int nEpochs = 30;
for( int epoch=0; epoch<nEpochs; epoch++ ){
    for(INDArray data : featuresTrain){
        net.fit(data,data);
    }
    System.out.println("Epoch " + epoch + " complete");
}
```

我们循环多轮训练数据。这里使用 API 时的一个值得注意的变化是下面一行：

```
net.fit(data,data);
```

这里数据也被用作网络的输出。这是因为使用自动编码器时，我们学习重建数据本身，所以数据既是输入又是输出，这使得调用 `fit()` 方法的方式与前面的示例稍有不同。

5.7.4 评估模型

当运行示例代码时，它为学得最好的图像生成图像，然后为重建误差最大的图像生成另一图像。图 5-2 是为学得最好的手写数字图像（例如重建误差最小的图像）生成的图像。

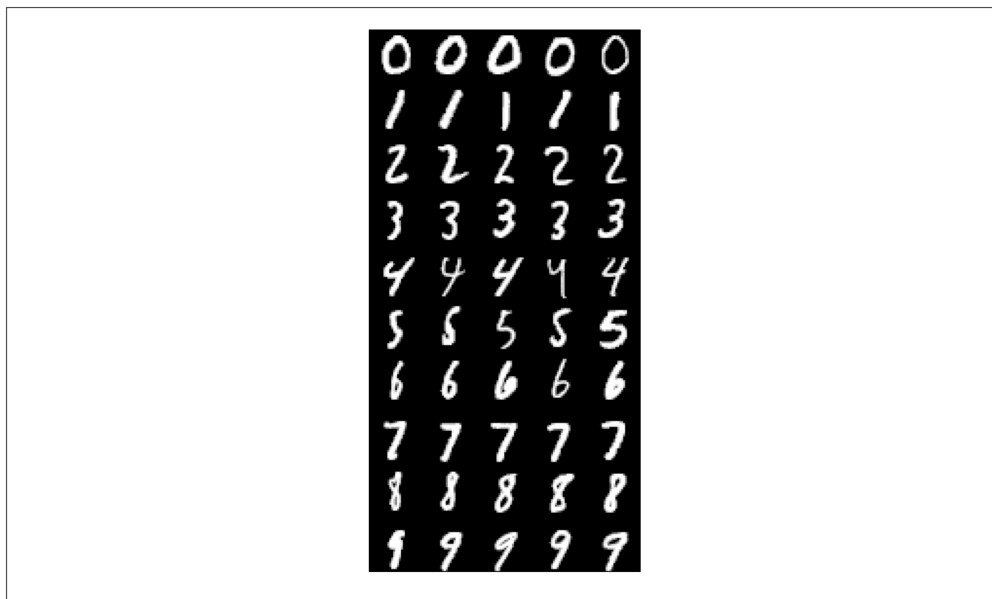


图 5-2：自动编码器学得最好的数字

重建误差最大的图像组成了另一幅图像，如图 5-3 所示。

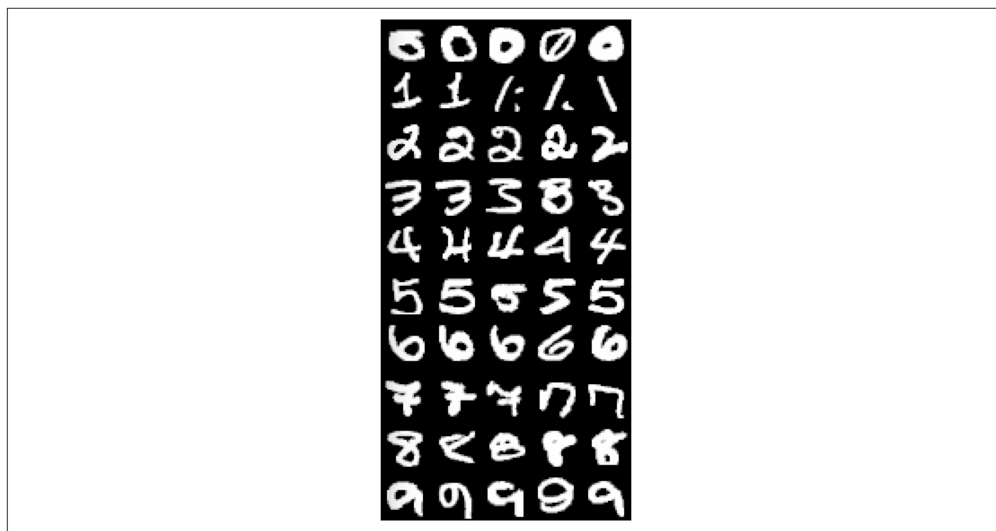


图 5-3：难以学习的手写数字

从图中可以看出，部分图像与其他图像相比肯定会被视作异常。我们并不总是知道数据有怎样的异常，但是如果把它建立在一个自动编码器这样的模型上，那么我们就有了一个不需要事先了解异常的工具。

5.8 使用变分自动编码器重建MNIST数字

第 3 章介绍了变分自动编码器（VAE）的概念，它是一种在无监督学习方式下重建输入数据的方法。

VAE 有很多用途，其中最常见的用途有：

- 无监督学习以及对特征的半监督学习，它的大体思想是学习大量未标记数据和少量已标记数据（当标记数据有限时，可以得到比单独使用标记数据更好的结果）；
- 异常检测（无监督方式）；
- 作为生成模型（例如生成样本），它们可以生成图像（如本例所示），也可以生成句子等。

这个示例（http://bit.ly/2tU4zT8）展示了如何生成 MNIST 数字数据集的变体，显示了 VAE 作为生成模型的能力。

5.8.1 重建MNIST数字的代码列表

下一个例子（示例 5-6）是在 MNIST 上训练 VAE，以展示 VAE 的生成能力。这个例子有意地在二维网格上增加了一个小的隐藏状态 Z（两个值），用于可视化。

经过训练，这个例子绘制了两幅图。

• MNIST 数字重建空间与潜在空间。
• 随着训练的进行（每 N 个小批量），MNIST 测试集的潜在空间值的变化。

注意这两幅图的顶部都有一个滑块。滑动滑块，观察重建空间和潜在空间如何随时间变化。

示例 5-6　用 Java 语言中的 VAE 对 MNIST 数字建模

```
public class VariationalAutoEncoderExample {
    private static final Logger log =
        LoggerFactory.getLogger(VariationalAutoEncoderExample.class);

    public static void main(String[] args) throws IOException {
        int minibatchSize = 128;
        int rngSeed = 12345;
        int nEpochs = 150;                    //总的训练轮数

        //绘图配置
        int plotEveryNMinibatches = 100;    //为后续的绘图而收集数据的频度
        double plotMin = -4;                //图形的最小值（x和y轴）
        double plotMax = 4;                 //图形的最大值（x和y轴）
        int plotNumSteps = 16;              //在plotMin和plotMax之间的重建的步数

        //用于训练的MNIST数据
        DataSetIterator trainIter = new MnistDataSetIterator(minibatchSize, true,
            rngSeed);

        //神经网络配置
        Nd4j.getRandom().setSeed(rngSeed);
        MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
            .seed(rngSeed)
            .iterations(1).optimizationAlgo(OptimizationAlgorithm
                 .STOCHASTIC_GRADIENT_DESCENT)
            .learningRate(1e-3)
            //.updater(Updater.RMSPROP).rmsDecay(0.95)
              .updater(updater.ADAM)
            .weightInit(WeightInit.XAVIER)
            .regularization(true).l2(1e-5)
            .list()
            .layer(0, new VariationalAutoencoder.Builder()
                .activation(Activation.LEAKYRELU)
                .encoderLayerSizes(512, 512)        //2个编码层，每层大小是256
                .decoderLayerSizes(512, 512)        //2个解码层，每层大小是256
                .pzxActivationFunction("identity")  //p(z|data) 激活函数
                .reconstructionDistribution(new BernoulliReconstructionDistribution(
                    //p(data|z)的伯努利分布（只有二元值或0到1的数据）
                    Activation.SIGMOID.getActivationFunction()))
                .nIn(28 * 28)                       //输入大小：28x28
                .nOut(2)                            //潜在变量空间的大小：p(z|x)。
                                                    //这里为了绘图方便起见，将其设
                                                    //为二维，通常维度会更多
                .build())
            .pretrain(true).backprop(false).build();

        MultiLayerNetwork net = new MultiLayerNetwork(conf);
        net.init();
```

```
    //获取变分自动编码器层
    org.deeplearning4j.nn.layers.variational.VariationalAutoencoder vae
        = (org.deeplearning4j.nn.layers.variational.VariationalAutoencoder)
            net.getLayer(0);


    //用于绘图的测试数据
    DataSet testdata = new MnistDataSetIterator(10 000, false, rngSeed).next();
    INDArray testFeatures = testdata.getFeatures();
    INDArray testLabels = testdata.getLabels();
    //plotMin和plotMax之间X/Y网格值
    INDArray latentSpaceGrid = getLatentSpaceGrid(plotMin, plotMax, plotNumSteps);

    //用于之后绘图而储存数据的列表
    List<INDArray> latentSpaceVsEpoch = new ArrayList<>(nEpochs + 1);
    //在训练开始之前，收集和记录潜在空间
    INDArray latentSpaceValues = vae.activate(testFeatures, false);
    latentSpaceVsEpoch.add(latentSpaceValues);
    List<INDArray> digitsGrid = new ArrayList<>();

    //执行训练
    int iterationCount = 0;
    for (int i = 0; i < nEpochs; i++) {
        log.info("Starting epoch {} of {}",(i+1),nEpochs);
        while (trainIter.hasNext()) {
            DataSet ds = trainIter.next();
            net.fit(ds);

            //对于每N=100个小批量：
            //(a) 为之后绘图而收集测试集的潜在空间
            //(b) 在网格的每一点收集重建值
            if (iterationCount++ % plotEveryNMinibatches == 0) {
                latentSpaceValues = vae.activate(testFeatures, false);
                latentSpaceVsEpoch.add(latentSpaceValues);

                INDArray out = vae.generateAtMeanGivenZ(latentSpaceGrid);
                digitsGrid.add(out);
            }
        }

        trainIter.reset();
    }

    //绘制MNIST测试集 – 潜在空间vs.迭代器（默认每100个小批量）
    PlotUtil.plotData(latentSpaceVsEpoch, testLabels, plotMin, plotMax,
        plotEveryNMinibatches);

    //绘制重建值 – 潜在空间vs.网格
    double imageScale = 2.0;        //增加/减少此值以放大数字
    PlotUtil.MNISTLatentSpaceVisualizer v =
        new PlotUtil.MNISTLatentSpaceVisualizer(imageScale, digitsGrid,
            plotEveryNMinibatches);
    v.visualize();
}
```

```
    //简单地为从x=plotMin到plotMax，以及从y=plotMin到plotMax的范围
    //返回二维网格：(x,y)
    private static INDArray getLatentSpaceGrid(double plotMin, double plotMax,
        int plotSteps) {
        INDArray data = Nd4j.create(plotSteps * plotSteps, 2);
        INDArray linspaceRow = Nd4j.linspace(plotMin, plotMax, plotSteps);
        for (int i = 0; i < plotSteps; i++) {
            data.get(NDArrayIndex.interval(i * plotSteps, (i + 1) * plotSteps),
                NDArrayIndex.point(0)).assign(linspaceRow);
            int yStart = plotSteps - i - 1;
            data.get(NDArrayIndex.interval(yStart * plotSteps,
                (yStart + 1) * plotSteps), NDArrayIndex.point(1)).assign(linspaceRow
                    .getDouble(i));
        }
        return data;
    }
}
```

下面解释生成的潜在空间网格和生成的 MNIST 图像。

5.8.2 VAE模型的检验

图 5-4 给出了由这个示例代码生成的图像。这些是在网络训练过程中，特定的迭代对潜在空间的重建。



潜在空间变量

在统计中，潜在空间变量通过数学模型推导或推断。这与观察到的变量不同。

不同的训练进行轮数和训练设置，可以生成不同的图像。

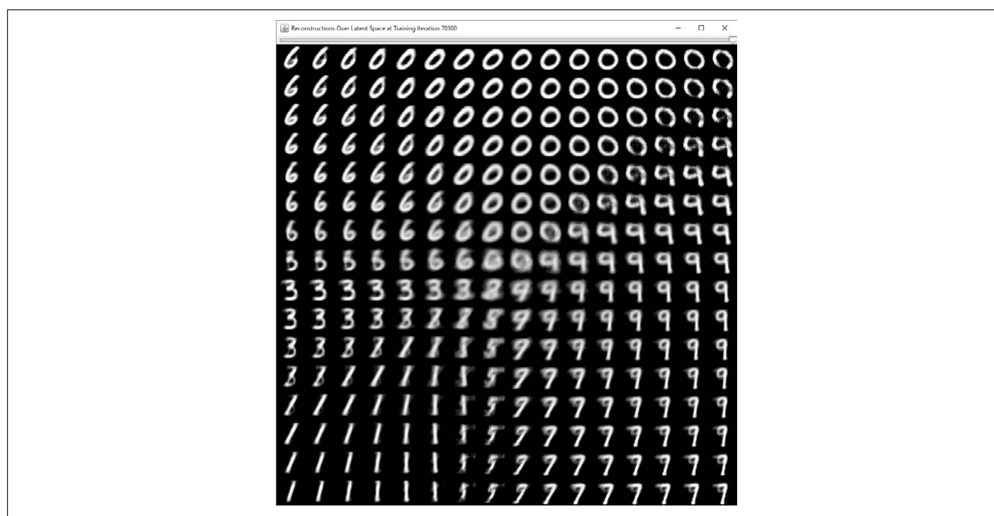


图 5-4：VAE 示例生成的 MNIST 数字

为了更好地理解这些图像如何生成，将这个示例与第 3 章介绍的 VAE 的网络架构联系起来。图 5-5 展示了 VAE 的网络架构。

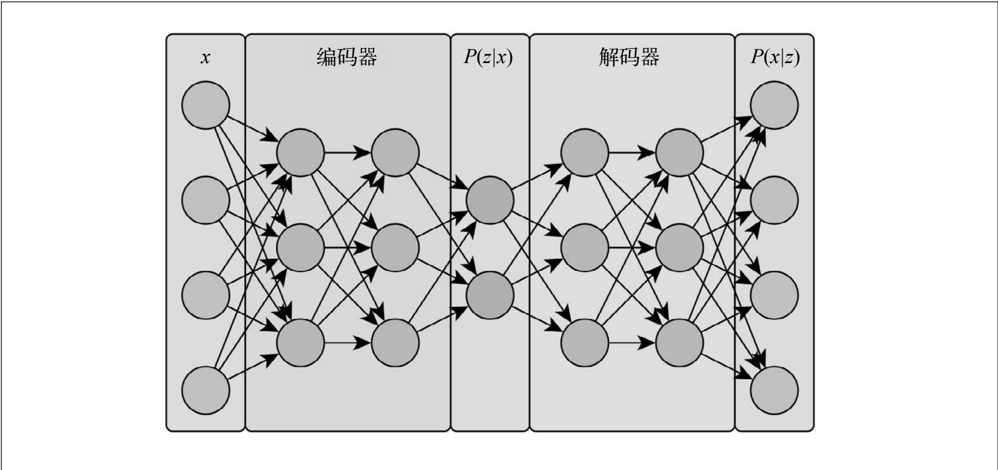


图 5-5：VAE 网络结构

在图 5-6 中，MNIST 测试数据的 VAE 潜在空间的散点图是图 5-5 所示的网络的编码器部分。

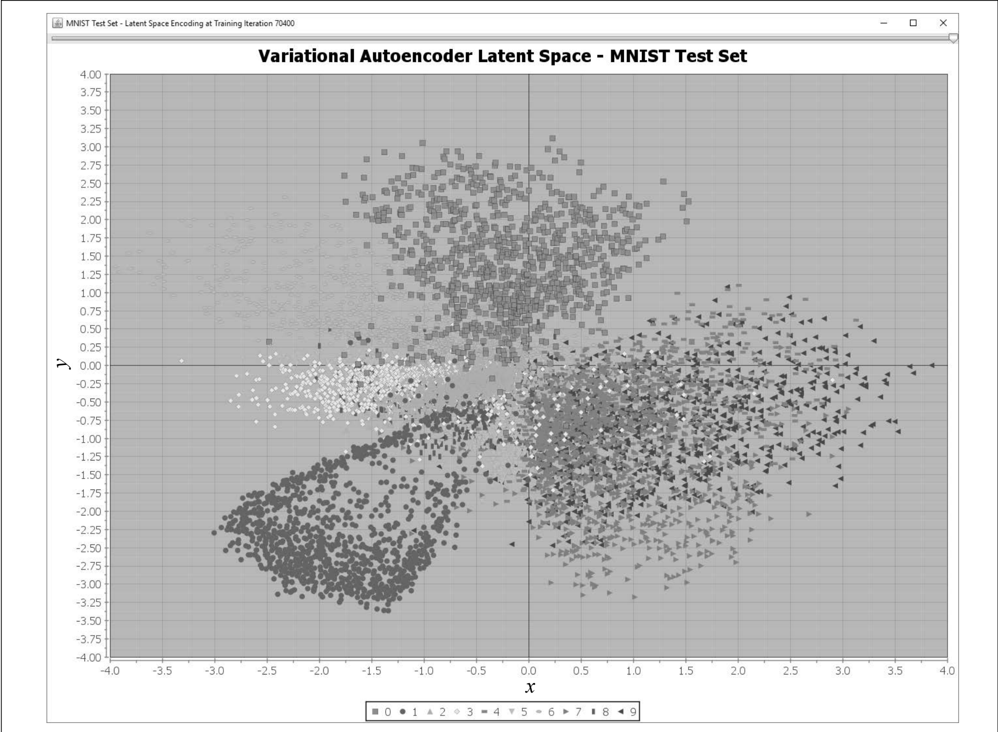


图 5-6：VAE 潜在空间的散点图与在 MNIST 测试数据集上训练轮数的关系

1. 理解散点图

图 5-6 中的散点图是可变的潜在空间在每一轮的投影，将 MNIST 测试集数据（如 x）投影到值 $p(z|x)$ 中，其中 $p(z|x)$ 是具有两个值的高斯分布，z 是潜在变量。

可以把 z 看作压缩自编码器中瓶颈的概率版本。在训练过程中，基本上从 z 中随机抽样以给出实际数值。显然，我们不能前向传递概率分布，但我们可以前向传递从该概率分布中抽取的数值（例如使用排序集上的蒙特·卡罗估计方法）。

对于建模，首先假设数据从 $p(z)$ 开始生成（例如潜在空间上的一些概率分布，并从中采样），然后假设有一些过程使用分布 $p(x|z)$ 来生成数据 x。

我们取 $p(z|x)$ 的平均值，并将其绘制在散点图上。通常有两个以上的值，但是我们只使用两个，因为想得到漂亮的图片。

2. 理解生成的图像

图 5-4 中生成的图像是图 5-6 中网络的右侧部分。具体说来，生成一个 16×16 的网格，值的范围是 −4 到 +4，然后将这些值前向传递，就好像它们是 $p(z|x)$ 的平均值一样。

为了以另一种方式理解渲染，只需要设置 z 值并通过解码器进行前向传递，以得到 $p(x|z)$（重建的 MNIST 数据）。

在这种情况下，$p(x|z)$ 是伯努利分布，在 0 到 1 范围内是有界的。这意味着可以只绘制这些概率（这也恰好是伯努利分布的平均值），其中 0 到 1 被转化为 0 到 255 的像素值。在这种情况下，x 和 $p(x|z)$ 的大小相同：输入 / 输出均为 784（来自 MNIST 的 28×28 大小的图像）。



散点图与生成图像的关系

从某种意义上说，散点图和生成的图像是从不同角度展开的同一事物。

5.9 深度学习在自然语言处理中的应用

在自然语言处理（NLP）领域，深度学习已被证明是高效的。这一领域的深度学习的常见应用包括词性标注、字符生成和学习词嵌入等技术。以下是本章将重点介绍的一些 NLP 应用程序。

- 使用 Word2Vec 进行学习词嵌入。
- 具有段落向量的句子的分布式表示。
- 基于段落向量的文档分类。

下面依次展示这些应用。

5.9.1 使用Word2Vec的学习词嵌入

Word2Vec 通过学习周围单词的上下文来检测单词之间在数学上的相似性。Word2Vec 创建

向量，这些向量是单词特征的分布式数字表示，例如单个词的上下文。Word2Vec 以文本语料库为输入数据来训练，并产生单词向量 / 词嵌入的列表作为输出模型。生成的词嵌入中单词含义和关系是空间编码的，并且具有向量计算等有用特性，稍后将介绍。

1. Word2Vec模型及算法

该算法首先从输入训练数据构建词汇表，然后构建单词的表示。我们不会像使用其他向量化技术那样一开始为每个文档构建向量。然而对于 Word2Vec 来说，基于输入语料库训练的输出数据集是语料库中唯一的单词集合，每个单词都带有一个向量。这个输出中的每个向量包含单词的上下文（或用法）。Word2Vec 使用前馈神经网络语言模型（NNLM）以无监督的学习方式构建这些上下文单词向量。在实践中，这是一个用 SGD 进行训练的两层神经网络。

单词向量代表一系列单词，允许对其上下文建模。它们代表一个单词相对于其周围单词的用法。当在命名实体识别（NER）、词性标注和语义角色标注（SRL）等场景中要对围绕单词的大量上下文信息进行分类时，这确实很方便。



生成的 Word2Vec 向量的特征数量

基于神经网络学到的单词的分布式表示结果，每个单词向量具有大约 50 到 300 个特征。

该领域的另一种技术是潜在语义索引（LSI），它检测那些看起来可以结合在一起的维度，并将其合并为单个的维度，这有助于加速聚类等计算。Word2Vec 的设计者考虑了许多估计单词连续表示的方法，包括潜在语义分析（LSA）和潜在狄利克雷分配（LDA）。Word2Vec 在保持单词之间的线性规则性方面明显优于 LSA，并且比 LDA 计算成本低。

2. 上下文建模

单词向量以多单词窗口的形式保留源文本中单词周围的上下文。基于机器学习的目的，将单词的含义定义为最接近它的单词。有了足够的数据，我们可以使用这些语义，并在语料库中构建词语的表示，从而相当准确地对一个单词建模。Word2Vec 从单词窗口创建特征，其中包括单个词的上下文。

当构建一个 Word2Vec 模型时，我们使用一个移动窗口一次跨越（通常）五个单词。通过移动窗口训练 Word2Vec 的概念在许多类似的技术中也有，如语义标注、SRL 和 NER。Word2Vec 使用**维特比算法**构建模型，根据给定的从一个状态迁移到另一个状态的概率或“迁移矩阵”计算最可能的事件序列（标签）。

3. 学习相似含义和语义关系

可以使用向量距离度量的方法找到相似的单词，比如欧几里得距离（第 1 章介绍过）或余弦距离。这使得我们可以得到余弦相似距离短的词，或者说，“相近的表示”。表 5-3 列出了一些基于 Word2Vec 模型训练的最接近“France”的单词的例子。

表5-3：Word2Vec余弦相似性

单　　词	余弦距离
Spain	0.678515
Belgium	0.665923
Netherlands	0.652428
Italy	0.633130
Switzerland	0.622323
Luxembourg	0.610033
Portugal	0.577154
Russia	0.571507
Germany	0.563291
Catalonia	0.534176



理解余弦距离

余弦距离 1.0 表示两个相同的词之间完美匹配（同一性，如“阿姆斯特丹”等于“阿姆斯特丹”）。余弦距离越接近值 1.0，两个词向量的意义越相似。

我们需要在向量维度足够大的大数据集上训练 Word2Vec 模型，以查看词向量空间中的强正则性。

4. 向量计算与词嵌入

还可以用 Word2Vec 向量进行一些基本运算，示例如下：

```
vector('Rome') = vector('Paris') - vector('France') + vector('Italy')
```

这个运算的结果是一个非常接近 vector('Rome') 的向量。其他有趣的相似性语义的例子有：“big”这个词与“bigger”相似，“small”与“smaller”也相似。一个有趣的练习是计算与 small 相似的词，就像“biggest”与“big”相似一样。我们通过词向量表示的简单代数运算计算。

```
vector('smallest') = vector("biggest") – vector("big") + vector("small")
```

接下来用 1NN 搜索（https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm）的方式搜索向量空间，计算词之间的距离（这里是余弦距离），寻找最接近搜索项的单词，得到词向量。如果词向量经过很好的训练，应该能通过这种技术找到正确的单词（smallest）。也可以发行一些细微的关系，比如国家的首都城市（例如法国的巴黎、德国的柏林）。

这些语义关系可以高效地改进现有的 NLP 技术，如机器翻译、信息检索和问答系统。下面来看一个使用 Word2Vec 的简单例子。

5. Word2Vec例子的Java代码

示例 5-7 中的代码（http://bit.ly/2txt1HL）从原始文本句子学习向量表示。这是展示学习语料库中词义的较大潜力的一个例子。

示例 5-7　Word2Vec 的代码示例

```
public class Word2VecRawTextExample {

    private static Logger log =
        LoggerFactory.getLogger(Word2VecRawTextExample.class);

    public static void main(String[] args) throws Exception {

        //获取文本文件的路径
        String filePath = new ClassPathResource("raw_sentences.txt").getFile()
            .getAbsolutePath();

        log.info("Load & Vectorize Sentences....");
        //去除每行前后的空白
        SentenceIterator iter = new BasicLineIterator(filePath);
        //基于空格将行分割为单词
        TokenizerFactory t = new DefaultTokenizerFactory();

        /*
            CommonPreprocessor对每个单词应用以下正则表达式：
            [\d\.:,"'\(\)\[\]|/?!;]+
            这样就能高效删除所有数字、标点符号和一些特殊符号
            另外，它还将所有单词一律变为小写。
         */
        t.setTokenPreProcessor(new CommonPreprocessor());

        log.info("Building model....");
        Word2Vec vec = new Word2Vec.Builder()
                .minWordFrequency(5)
                .iterations(1)
                .layerSize(100)
                .seed(42)
                .windowSize(5)
                .iterate(iter)
                .tokenizerFactory(t)
                .build();

        log.info("Fitting Word2Vec model....");
        vec.fit();

        log.info("Writing word vectors to text file....");

        //将词向量写入文件
        WordVectorSerializer.writeWordVectors(vec, "pathToWriteto.txt");

        //打印出最接近单词"day"的10个单词。这是一个演示如何处理这些词向量的例子。
        log.info("Closest Words:");
        Collection<String> lst = vec.wordsNearest("day", 10);
        System.out.println("10 Words closest to 'day': " + lst);

    }
}
```

下面讨论这段代码中特定的部分，以便你更好地理解示例中发生的事情。

6. 理解Word2Vec例子

在下面的代码片段中，可以看到如何使用 `SentenceIterator` 对象引用和加载代码示例库中包含的原始句子文件。

```
//获取文本文件的路径
String filePath = new ClassPathResource("raw_sentences.txt").getFile()
    .getAbsolutePath();

log.info("Load & Vectorize Sentences....");
//去除每行前后的空白
SentenceIterator iter = new BasicLineIterator(filePath);
//基于空格将行分割为单词
TokenizerFactory t = new DefaultTokenizerFactory();

/*
    CommonPreprocessor对每个单词应用以下正则表达式：
    [\d\.:,"'\(\)\[\]|/?!;]+
    这样就能高效地删除所有数字、标点符号和一些特殊符号
    另外，它还将所有单词一律变为小写。
 */
t.setTokenPreProcessor(new CommonPreprocessor());
```

这里，预处理使用 `CommonPreprocessor` 对象来执行。这个例子中大多数文本操作都用这些类在底层执行，使这个例子更简单。

Word2Vec 的架构与本章前面介绍的网络架构看起来不同，这是因为 `Word2Vec` 类在 DL4J 中基本上是独立的。

```
log.info("Building model....");
Word2Vec vec = new Word2Vec.Builder()
         .minWordFrequency(5)
         .iterations(1)
         .layerSize(100)
         .seed(42)
         .windowSize(5)
         .iterate(iter)
         .tokenizerFactory(t)
         .build();
```

你可能还注意到在网络架构中直接设置了句子迭代器，这不是功能上的主要区别，而更多是由遗留在这个特定网络上过去的 API 设计导致的。

鉴于为这个网络架构设置了句子迭代器，那么只需简单地调用 `.fit()` 方法，而不需要其他参数来训练网络，如下面的代码片段所示：

```
log.info("Fitting Word2Vec model....");
vec.fit();
```

在 `.fit()` 方法完成后，这个例子保存网络并打印出 10 个最接近单词“day”的单词。

7. Word2Vec的其他实际应用

Word2Vec 神经网络的输出是一个词汇表，其中每项都有一个附属向量，它可以被输入到深度学习网络，或者简单地用于查询以检测词之间的关系。也可以使用 Word2Vec 向量来对文档进行分类，本章稍后将介绍它（使用段落向量）。

为了进行规范文档分类，从语料库中生成单词向量之后，将要分类的文档中出现的每个单词的向量加到一起。这样就得到了一个表示文档的向量，和包含在每个单词向量中的上下文信息。之后我们以和基于文档生成 TF-IDF 向量同样的方式，使用这个文档向量。

> **使用 GloVe 学习表示单词的全局向量**
>
> Word2Vec 的一种替代方法是 GloVe。
>
> GloVe 是一种构建词嵌入的无监督学习算法。它基于从输入数据（例如文本文档组）聚合得到的全局中单词与单词共同出现的统计数据来构建模型。二者的主要区别是：Word2Vec 是一个“预测”模型，而 GloVe 是一个“基于计数”的模型。值得注意的是，GloVe 被认为比 Word2Vec 更难训练和调优。
>
> DL4J 有一个与前面 Word2Vec 练习类似的 GloVe 示例（http://bit.ly/2sNV7S9，你可以在示例存存库 https://github.com/deeplearning4j/oreilly-book-dl4j-examples/）中找到它。

5.9.2 具有段落向量的句子的分布式表示

如果继续基于 Word2Vec 中确立的思想，我们可以扩展这种技术，以相似的方式对任意更大的序列（例如句子或文档）建模。在这个例子中，我们将创建段落向量（例如 Doc2Vec）。

历史上，机器学习算法要求建模所用文档的表示长度是固定的，一个最常用的实现了这一目标的方法是词袋向量化方法。词袋法在向量中为文档中每个不同的词创建计数。不过这个方法有两个主要问题。

- 单词的顺序信息丢失。
- 没有对单词的语义建模。

使用段落向量，我们得到了表示可变长度文本序列（例如句子、段落或文档）的固定长度向量。每条逻辑记录由密集向量表示。研究表明段落向量法优于词袋法和其他文本表示法，也提高了对多种文本分类和情感分析问题的准确性。

这个例子使用 DL4J 中 Doc2Vec 实现的段落向量。Doc2Vec 是 Word2Vec 的扩展，它学着将标签和单词关联，而不是将单词与其他单词关联。Doc2Vec 以无监督的训练方法学习输入语料库，因为不需要明确地给它标签。接下来是一个使用 DL4J 对段落向量建模的代码示例。

1. 建立段落向量

示例 5-8 的代码（http://bit.ly/2tPQeHM）构建了在训练语料库中存在的所有句子的分布式表示（段落向量）。示例的训练数据包含在示例代码库中，你只需要克隆代码库，在本地运行这个类即可。

示例 5-8　在 Java 中使用 DL4J 构建段落向量

```
public class ParagraphVectorsTextExample {

    private static final Logger log =
        LoggerFactory.getLogger(ParagraphVectorsTextExample.class);

    public static void main(String[] args) throws Exception {
        ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");
        File file = resource.getFile();
        SentenceIterator iter = new BasicLineIterator(file);

        AbstractCache<VocabWord> cache = new AbstractCache<>();

        TokenizerFactory t = new DefaultTokenizerFactory();
        t.setTokenPreProcessor(new CommonPreprocessor());

        /*
            如果没有LabelAwareIterator，可以使用同步标签生成器，它会用它自己的
            标签来标记每个文档/序列/行。但是如果已经准备好了LabelAwareIterator，
            可以把它提供给内部标签。
        */
        LabelsSource source = new LabelsSource("DOC_");

        ParagraphVectors vec = new ParagraphVectors.Builder()
                .minWordFrequency(1)
                .iterations(5)
                .epochs(1)
                .layerSize(100)
                .learningRate(0.025)
                .labelsSource(source)
                .windowSize(5)
                .iterate(iter)
                .trainWordVectors(false)
                .vocabCache(cache)
                .tokenizerFactory(t)
                .sampling(0)
                .build();

        vec.fit();

        /*
            训练语料库中，很少有包含非常接近的单词的行。下面这些句子在向量空间中应
            该非常接近
            3721行：This is my way .
            6348行：This is my case .
            9836行：This is my house .
            12493行：This is my world .
            16393行：This is my work .
            这是一个特殊的句子，与前面的句子没有共同之处。
            9853行： We now have one .
            注意文档从0开始索引
         */
```

```
        double similarity1 = vec.similarity("DOC_9835", "DOC_12492");
        log.info("9836/12493 ('This is my house .'/'This is my world .') similarity:
            " + similarity1);

        double similarity2 = vec.similarity("DOC_3720", "DOC_16392");
        log.info("3721/16393 ('This is my way .'/'This is my work .') similarity:
           " + similarity2);

        double similarity3 = vec.similarity("DOC_6347", "DOC_3720");
        log.info("6348/3721 ('This is my case .'/'This is my way .') similarity:
           " + similarity3);

        //这种情况下可能性应该显著降低。
        double similarityX = vec.similarity("DOC_3720", "DOC_9852");
        log.info("3721/9853 ('This is my way .'/'We now have one .') similarity: "
            + similarityX +"(should be significantly lower)");
    }
}
```

正如在结果中所看到的，前三个相似性查询比最后一个的得分更高，最后一个的得分显著降低。

2. 理解段落向量的示例

段落向量的示例在结构上与前面的 Word2Vec 示例相似，它们加载输入训练数据、建立网络，然后训练数据的方式相同。同样，文件和目录处理由引入的帮助类自动完成。

```
ClassPathResource resource = new ClassPathResource("/raw_sentences.txt");
File file = resource.getFile();
SentenceIterator iter = new BasicLineIterator(file);

AbstractCache<VocabWord> cache = new AbstractCache<>();

TokenizerFactory t = new DefaultTokenizerFactory();
t.setTokenPreProcessor(new CommonPreprocessor());

/*
    如果没有LabelAwareIterator，可以使用同步标签生成器，它会使用它自己的标签来
    标记每个文档/序列/行。但是如果已经准备好了LabelAwareIterator，可以把它提供
    给内部标签。
*/
LabelsSource source = new LabelsSource("DOC_");
```

DL4J 中的段落向量实现使用了一个被称为 ParagraphVectors 的帮助类，如下所示：

```
ParagraphVectors vec = new ParagraphVectors.Builder()
        .minWordFrequency(1)
        .iterations(5)
        .epochs(1)
        .layerSize(100)
        .learningRate(0.025)
        .labelsSource(source)
        .windowSize(5)
```

```
            .iterate(iter)
            .trainWordVectors(false)
            .vocabCache(cache)
            .tokenizerFactory(t)
            .sampling(0)
            .build();

    vec.fit();
```

关键的超参数直接使用此类中的方法设置，因为与其他使用 `MultiLayerConfiguration` 类的网络更广泛的方法相比，`ParagraphVectors` 具有更专用的 API（像 Word2Vec）。

同样，我们调用 `.fit()` 方法。在示例的最后，一些文档通过余弦相似性度量方法（如在 Word2Vec 示例中看到的）相互比较。这个例子有助于你更好地理解如何使用 DL4J 构建段落向量。下一个例子将展示如何将段落向量应用到更广泛的程序中，使用它来进行分类。



生成序列向量

段落向量是**序列向量**的一种实现。DL4J 项目中有 Skip-Gram 模型的通用实现，它是 Word2Vec、段落向量和 DeepWalk 等模型的基础。

Hinton 关于段落向量潜在意义的演讲

2015 年，在伦敦皇家学会的一次演讲中，Geoffrey Hinton 讲到：

> 这对于文档处理意义重大。如果我们将一个句子转换为一个能捕获句子含义的向量，那么谷歌搜索可以做得更好，可以基于文档的含义进行搜索。
>
> 此外，如果能够把一个文档中的每个句子转换成向量，那么你就可以使用那个向量序列，尝试对自然推理建模。这是老式人工智能永远无法做到的。
>
> 如果能够阅读网络上的每个英文文档，并将每个句子转换成思维向量，那么你就拥有大量的数据，可以像人一样训练推理系统。
>
> 现在你可能不希望它像人一样推理，但至少我们可以知道它们会想什么。
>
> 我认为在未来几年，这种把句子转化为思维向量的能力将迅速改变我们对文档理解的程度。
>
> 要想达到像人一样理解文档的程度，我们可能需要有与人同样级别的资源。我们的大脑里有数万亿个连接，但迄今为止我们建立的最大网络只有数十亿个连接。因此我们有几个数量级的差距，但我相信搞硬件的人会解决这个问题。

5.9.3 使用段落向量进行文档分类

正如本章前面提到的，我们可以在 NLP 和文档分类的上下文中使用段落向量。这个示例将在应用程序中使用与上一个示例中相似的段落向量模型来构建文档分类器，这个分类器会给出类似于下面这样含有三个标签的输出。

```
Document 'health' falls into the following categories:
    health: 0.29721372296220205
    science: 0.011684473733853906
    finance: -0.14755302887323793
```

从上面的片断可以看出，得分最高的标签就是分类。下面是示例 5-9 的段落向量分类应用程序（http://bit.ly/2sOpJ5Q）。

示例 5-9　在 Java 中使用段落向量对文档进行分类

```
public class ParagraphVectorsClassifierExample {

    ParagraphVectors paragraphVectors;
    LabelAwareIterator iterator;
    TokenizerFactory tokenizerFactory;

    private static final Logger log =
        LoggerFactory.getLogger(ParagraphVectorsClassifierExample.class);

    public static void main(String[] args) throws Exception {

      ParagraphVectorsClassifierExample app =
         new ParagraphVectorsClassifierExample();
      app.makeParagraphVectors();
      app.checkUnlabeledData();
        /*
                你的输出应该是这样的：
                Document 'health' falls into the following categories:
                    health: 0.29721372296220205
                    science: 0.011684473733853906
                    finance: -0.14755302887323793
                Document 'finance' falls into the following categories:
                    health: -0.17290237675941766
                    science: -0.09579267574606627
                    finance: 0.4460859189453788
                    所以，现在我们知道了还没有读过的文档的类别。
         */
    }

    void makeParagraphVectors()  throws Exception {
      ClassPathResource resource = new ClassPathResource("paravec/labeled");

      //为数据集构建迭代器
      iterator = new FileLabelAwareIterator.Builder()
              .addSourceFolder(resource.getFile())
              .build();

      tokenizerFactory = new DefaultTokenizerFactory();
      tokenizerFactory.setTokenPreProcessor(new CommonPreprocessor());

      //ParagraphVectors训练配置
      paragraphVectors = new ParagraphVectors.Builder()
              .learningRate(0.025)
              .minLearningRate(0.001)
              .batchSize(1000)
```

```
                .epochs(20)
                .iterate(iterator)
                .trainWordVectors(true)
                .tokenizerFactory(tokenizerFactory)
                .build();

        //开始训练模型
        paragraphVectors.fit();
      }

      void checkUnlabeledData() throws FileNotFoundException {
        /*
         在这个点上，假设模型已经建立了，并且可以检查未标记文档属于哪些类别。
         因此开始加载未标记的文档并检查它们。
       */
       ClassPathResource unClassifiedResource =
         new ClassPathResource("paravec/unlabeled");
       FileLabelAwareIterator unClassifiedIterator = new FileLabelAwareIterator
         .Builder()
               .addSourceFolder(unClassifiedResource.getFile())
               .build();

       /*
        现在将对未标记的数据进行迭代，并检查它可以被分配哪个标签。
        请注意：对于许多领域来说，一个文档同时落入几个标签是正常的，
        并且文档在每个标签中都有不同的“权重”。
       */
       MeansBuilder meansBuilder = new MeansBuilder(
           (InMemoryLookupTable<VocabWord>)paragraphVectors.getLookupTable(),
             tokenizerFactory);
       LabelSeeker seeker = new LabelSeeker(iterator.getLabelsSource().getLabels(),
           (InMemoryLookupTable<VocabWord>) paragraphVectors.getLookupTable());

       while (unClassifiedIterator.hasNextDocument()) {
           LabelledDocument document = unClassifiedIterator.nextDocument();
           INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
           List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);

           /*
            请注意，document.getLabel()仅表示现在正在查看的文档，这里不打印整个文档名称。
            所以，这两个文档的标签就像标题一样，只是为了适当地可视化分类。
           */
           log.info("Document '" + document.getLabel()
               + "' falls into the following categories: ");
           for (Pair<String, Double> score: scores) {
               log.info("        " + score.getFirst() + ": " + score.getSecond());
           }
       }

      }
    }
```

示例 5-9 的整体思路与我们在 LDA（例如“主题空间建模”）中对 `ParagraphVectors` 类的使用方式相同。

这个例子假设训练所用的数据中有一些已标记类别的，还有一些未标记的文档。应用程序的目标是通过利用段落向量中所包含的信息来确定这些未标记的文档属于哪个类别。

1. 理解段落向量分类的示例

这个例子和前面的例子的一些做法类似，但它扩展到了对未标记的文档进行分类。主要的程序代码如下所示：

```
public static void main(String[] args) throws Exception {

  ParagraphVectorsClassifierExample app =
    new ParagraphVectorsClassifierExample();
  app.makeParagraphVectors();
  app.checkUnlabeledData();
```

可以看到如何用 `.makeParagraphVectors()` 方法封装前面的例子的。下面继续使用这个段落向量模型对新的未标记文档进行分类。在 `.checkUnlabledData()` 方法中，代码循环对文档列表进行分类：

```
LabelledDocument document = unClassifiedIterator.nextDocument();
INDArray documentAsCentroid = meansBuilder.documentAsVector(document);
List<Pair<String, Double>> scores = seeker.getScores(documentAsCentroid);
```

下面的代码将生成的分数打印到屏幕上：

```
log.info("Document '" + document.getLabel() + "'
    falls into the following categories: ");
for (Pair<String, Double> score: scores) {
    log.info("        " + score.getFirst() + ": " + score.getSecond());
}
```

它将产生以下输出：

```
Document 'health' falls into the following categories:
    health: 0.29721372296220205
    science: 0.011684473733853906
    finance: -0.14755302887323793
```

最高分就被视作分类。

```
health: 0.29721372296220205
```

在前面的输出中可以清楚地看到文档被分类为“health”，这是正确的，因为它与数据中的相关标签匹配。

2. Word2Vec方法的进一步探索

正如之前关于 CNN 的章节提到的，这里介绍的针对不同架构的许多应用程序仅仅是很好的起点。通常而言，嵌入指由神经网络学到的数据关系的映射。Word2Vec 方法的其他有趣变体包括以下领域的应用：

- 扩展到特定领域
- 图分析
- 推荐

• 图像识别

扩展到特定领域——Gov2Vec。Word2Vec 的一个有趣的应用是扩展到政府法律文本分析领域的、被称为"Gov2Vec"的应用。下面是作者论文中的描述。

> 我们将每个机构的整个法律语料库的表示和所有语料库共享的词汇嵌入连续的向量空间，来比较不同机构间的政策差异。我们将 Gov2Vec 方法应用于最高法院的意见、总统的行动和国会法案的官方摘要。

这种神经单词嵌入的变体能够理解国会和总统可能批准 / 否决法案之间的关系，以及随着时间的推移，这种关系会如何演变。它们能够回答以下问题。

> 奥巴马和第 113 届众议院在应对气候变化方面有何不同？从环境或经济角度看有何区别？

随着深度学习的演化，向量计算方法最有可能更频繁出现。

图和 Node2Vec。在图分析领域，Node2Vec 是一个学习网络中节点的连续特征表示的算法框架。Node2Vec 的一个有趣的特性是它可以扩展到非常大的图（数百万个节点 / 边或更多）。

推荐引擎和 Item2Vec。这个概念的另一个有趣的应用是 Item2Vec，该方法用于推荐系统。在论文中，作者展示了神经网络词嵌入方法作为基于项目的协同过滤技术用于推荐。

计算机视觉和 FaceNet。在图像识别领域，有一个叫作 FaceNet 的应用，它用于生成面部特征（图像）表示的神经网络嵌入。

总的说来，Word2Vec 方法是一种使用嵌入来映射输入数据之间关系的有趣方法，它使我们能够进一步地对数据进行聚类、分类和比较。

第 6 章

深度网络调优

任何物质都是毒药，没有什么东西是无毒的，剂量决定一切。

——帕拉塞尔苏斯，15 世纪文艺复兴时期的医生、
植物学家、炼金术士、占星家和术士

6.1 深度网络调优的基本概念

本章将研究训练神经网络的方法和策略，主要关注以下方面：

- 与当前问题相匹配的网络架构；
- 超参数调优基础；
- 更好地理解学习过程。

显然，本章不会覆盖深度学习领域中已知调优工作的方方面面，我们将介绍最相关的材料，让你了解深度架构调优的核心概念。第 7 章将重点介绍对深度网络中最著名的架构进行调优的技术：

- DBN
- CNN
- RNN

目的不同，构建的神经网络也不同，我们从总体想法和思路开始吧。



对 DBN 调优

本章会在与 DBN 调优相关的部分介绍 RBM 的调优内容。

6.1.1 建立深度网络的思路

当开始行动前，你应该问自己两个问题。

- 我将对怎样的输入数据建模？
- 在模型被构建之后，我想得到怎样的输出？

理解正在建模的数据类型在很大程度上将决定你需要实现哪类深度学习的架构和输入层。从某个类别的数据中确定你想知道的内容，这将告诉你需要获得类别分数（用概率对事物进行分类）还是需要产生实值目标输出（回归），这也将决定你需要使用怎样的输出层。深度网络的许多细节都有变化和注意事项，但是这两个问题将在构建深度网络架构时帮助你建立一个设计良好的基础。当理解了这两点之后，你可以转到下一层的设计决策，其重点是设置参数本身：

- 层数
- 每层的参数数量

我们还需要基于参数数量，考虑特定网络架构的内存需求。层数和每层的参数数量将表示最终网络能够代表的数据中结构的容量。对于一些问题，可以用相对较少的参数（总神经元数量）表示一个特别复杂的模型。

确定了网络架构和层级之后，还需要考虑一些其他要素，包括：

- 权重初始化策略
- 激活函数
- 损失函数
- 优化算法
- 小批量
- 正则化

权重初始化策略通常取决于网络架构的类型以及输入数据的类型。能否很好地初始化权重会帮助或者阻碍学习过程。每层都需要一个激活函数来对输入和输出数据之间的非线性关系建模。激活函数有助于学习某些类型的特性，以及设置输出层以提供回归、分类等的答案。特定的层类型（激活函数类型）应与特定架构情况下的特定损失函数相匹配。



通过损失函数影响学习

损失函数用于定义你想让网络如何学习（例如分类或回归）。需要用合适的激活函数以及数据 / 标签类型来匹配损失函数。

根据问题的领域，还需要考虑不同类型的优化方法。

正则化方法有助于防止模型过于关注数据中的噪声，并保持权重尽可能的小，以便泛化到整个数据总体，例如对那些在训练期间未曾见过的样本。在本章你会看到，前面提到的许多设计决策是联系在一起的，或者对网络其他所有架构的决策都有影响。本章旨在解释这些设计的相互依赖性是如何起作用的。下面将这些高层次思想的部分内容总结为更清楚的步骤形式，你可以将它用作构建网络架构的指南。

6.1.2 构建思路的步骤

前一节从整体上讨论了构建一个深度网络架构的过程，这些过程总结为下面的步骤，你可以将这个指南用于大多数神经网络建模问题。

(1) 确定输入数据是什么：
  输入数据类型与网络架构相关。
(2) 确定预期结果是什么：
  a. 引导我们配置架构；
  b. 确定输出层类型。
(3) 建立网络架构，并考虑以下问题：
  a. 模型、结构和成本函数的选择都很重要；
  b. 根据架构选择一些隐藏层；
  c. 根据网络整体架构和特定层的目标选择每层的激活函数。
(4) 对训练数据执行以下任务：
  a. 数据清洗；
  b. 结果可视化；
  c. 执行向量化和规范化；
  d. 平衡类别（如有必要）；
  e. 创建分别用于测试、训练和验证的数据分片。
(5) 考虑具有平衡子数据集的超参数调整策略：
  增加子数据集的大小并根据需要调整超参数。
(6) 如果最终训练数据集很大，那么使用 Spark 更快地训练更多数据（在适当的情况下）。

这些步骤在特定的调优细节方面仍然不够具体，但是它们为我们构建任何深度网络提供了通常的步骤。当本章结束时，希望你会掌握一套很好的基本调优原则，以通向当今数据科学的新兴领域。接下来深入研究深度网络调优的细节，第一步是匹配输入数据与网络架构。

6.2 匹配输入数据与网络架构

正如前一节所讨论的，应当从考虑源数据集所表示的内容来开始深度网络设计过程，如下所示：

- 列式数据
- 图像数据
- 音频数据
- 视频数据
- 时间序列数据

列式或逗号分隔值（CSV）数据，通常是从关系数据库管理系统（RDBMS）导出的表或联接在一起的多组数据的结果非规范化的表，如下所示：

```
M,0.455,0.365,0.095,0.514,0.2245,0.101,0.15,15
M,0.35,0.265,0.09,0.2255,0.0995,0.0485,0.07,7
F,0.53,0.42,0.135,0.677,0.2565,0.1415,0.21,9
```

```
M,0.44,0.365,0.125,0.516,0.2155,0.114,0.155,10
I,0.33,0.255,0.08,0.205,0.0895,0.0395,0.055,7
I,0.425,0.3,0.095,0.3515,0.141,0.0775,0.12,8
```

这种数据既没有可提取特征的图像像素，也没有需要处理的时间依赖（时间序列数据），因此网络架构可以相对简单。这里建议你从一个简单的多层感知器神经网络开始。

对于图像分类任务，我们会使用 CNN。近年来 CNN 已被证明最擅长图像处理任务（相关更多信息，请参阅第 4 章）。

序列数据是任意模型中有一系列输入的情况。常见的序列数据是由服务器或传感器生成的日志数据，时间序列数据也是其中一种。我们将时间序列数据定义为具有一系列值的数据，每个数据都带有时间值。这些值按时间顺序排列，用于跟踪随着时间推移的实例的活动。在大多数机器学习模型中，需要基于每组时间序列数据提取一些特征，来为训练算法生成单个输入向量。对于任何序列或时间序列数据，应该考虑使用循环神经网络，这种神经网络能对 N 个输入向量建模，而不限于单个输入向量（参见第 4 章），这使得我们能够对随时间推移的活动建模。

循环神经网络通常在音频输入数据上表现良好。音频数据具有时间特性，一系列音频样本的时间序列能够产生波形，使得它非常适合用循环神经网络进行训练。

对于视频数据，需要使用更精细的计算技术从视频的一系列图像中提取答案。解决这个问题的一种方法是结合卷积、最大池化、密集（前馈）和循环（LSTM）层来对视频的每一帧进行分类。另一种方法是将视频的每一帧提取到图像文件中，然后使用 CNN 分析每一帧。



视频数据处理

在实践中，使用视觉流预处理能起很大的作用，它有助于捕获一些在单个帧中不存在的、数据中的短期时间关系（例如连续帧之间的移动）。

小结

对于特定的数据输入类型，推荐使用相应的网络架构。表 6-1 对这些建议做了清晰的总结。

表6-1：匹配输入类型与网络架构

输入数据类型	建议架构
列式数据，即 CSV 数据	多层感知器
图像	CNN
序列数据	RNN，尤其是 LSTM
音频	RNN，尤其是 LSTM
视频	CNN+RNN 混合结构



网络架构的旋转门

尽管对每个数据类型，使用推荐的网络架构可能工作得很好，但是请继续关注正在进行的大量新研究。近来网络架构的新变体发布的速度很快，所以可以以给出的建议为好的起点，但也要关注那些对这些架构的研究。

稍后将详细探讨网络架构，研究如何设置层和神经元的数量。

6.3 模型目标与输出层的关系

前面讨论了将输入数据类型与特定神经网络架构相匹配的话题，是时候检查输出层了，它涉及更多的考虑，因为它关系到我们期望从生成的模型中得到怎样的答案。

每层都有一个相应的激活函数，用于将信息传递到下一层。如果我们想从网络中获得输出，最后一层会被称为“输出层”，并根据预期的输出或答案（例如分类或回归，稍后介绍）设置其激活函数的类型。

6.3.1 回归模型的输出层

回归模型生成一个实值的输出，比如根据面积计算房子的价格。回归模型输出层的两个主要考虑因素是损失函数和激活函数。

- 损失函数

  对于回归输出层的损失函数，我们有多个有效的选择。最常见的是均方误差（MSE）或误差的平方和（L2）。

- 激活函数

  在这种情况下，我们使用恒等（线性）输出函数。

回归输出层及其他边界情况

有时回归输出层使用 tanh 激活函数（仅当所有数据被保证在 [−1, 1] 范围内），或者数据被保证在 [0, ∞) 范围内时使用 softplus 或修正线性的变体（leaky ReLU、随机 leaky ReLU）。

当标签数据在 [0, ∞) 范围内，在回归中使用 ReLU 会怎样？虽然激活函数本身可以产生正确的输出值范围，也同样是 [0, ∞)，但是还有一个问题需要注意：所谓的“死亡 ReLU”问题。

本质上，ReLU 可能陷入激活函数的输出为 0 的部分。如果发生这种情况，无论网络输入的值是多少，ReLU 的输出（网络预测）将总为 0。其他的 ReLU 变体（leaky 和随机 leaky）和 softplus 则不存在这个问题。

6.3.2 分类模型的输出层

在分类模型中，输出层中有 N 个输出单元，并为每个输出单元生成一个类别分数。如果 $N = 1$，模型就带有单个标签。在这种情况下，我们将对条件是否满足进行分类（例如垃圾邮件和非垃圾邮件）。如果 $N > 1$，那么对每个类别的输入进行评分，并且使用不同的输出层配置。这种情况下，可以把一个文档分类为：体育、商业、政治。也有文档属于多个类别的情况，例如体育和商业。

1. 单标签分类模型

在基本的单标签示例或者二元分类器中，对输出层使用 sigmoid 激活函数会得到单个输出，它的值介于 0.0 到 1.0 之间，表示文档是否确实是垃圾邮件。

当处理单个输出标签（对于两个类别，有 0.0 到 1.0 范围内的单个值）时，我们对输出层使用交叉熵损失函数。

> **二元分类器的输出是一个还是两个**
>
> 有一种情况：二元分类器使用 softmax 激活函数，输出层有两个输出。在这种情况下，输出将是和为 1 的两个值，较大的值是标签的索引。对于这种情况，我们使用 MCXENT 损失函数配合 softmax。
>
> 围绕对二元分类模型建模时哪种输出层方案最好，目前还存在争议。在数学上，使用 sigmoid 单输出与使用具有两个输出单元的 softmax 输出层相同（使用 MCXENT/ 负对数似然，以及独热表示方式的 [1, 0] 或 [0, 1]，而不是 0 或 1）。

2. 具有两个以上标签的模型

当有两个以上标签时，需要考虑两种情况。

- 有时，对于多个标签，我们希望模型给出最可能的那个标签。这被称为“多类别分类”。
- 其他时候，对于多个标签，我们希望每个输出带有多个标签（例如“人 + 车”），这被称为“多标签分类”。

下面来了解每种情况。

多类别分类模型。回顾一下第 2 章的讨论，如果有一个多分类建模问题，但我们只关心这些类别中的最高得分，那么对输出层使用 softmax 激活函数。在这个场景中，输出单元的数量等于要分类的类别数量。

输出层所有单元的输出总和为 1.0。假设输出是每个类别的概率，可能每个输出值不会正好等于 1.0 或 0.0，而是 0.0 到 1.0 之间的某个值。我们选择概率最高的类别（输出单元）作为分类结果，使用 `arg-max()` 函数来获得得分（预测类别）最高的类别的索引。

下面是在神经网络的输出层上设置了 softmax 激活函数的代码片段。

```
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
        .weightInit(WeightInit.XAVIER)
        .activation(Activation.SOFTMAX)
        .nIn(numHiddenNodes).nOut(numOutputs).build())
```

这个示例中特定的输出层部分有用 `.activation()` 方法指定的层类型。

大多数时候，负对数似然与 softmax 激活函数一起使用，如前面的代码片段所示。有时多分类交叉熵的用法与之类似，但它相当于负对数似然。

数据表示应该是输出的“独热”编码。这与建立输入 / 输出训练向量有关，需要记住将输出向量转换为独热表示。DL4J 的 `RecordReaderDataSetIterator` 可以帮我们自动将分类索

引（0 到 [类别数 -1]）转换为独热向量。



处理大量的标签

对于目标标签数量达到一万个的情况，可以使用分级 softmax 输出层。分级 softmax 是 softmax（更）快速的逼近，适合有大量类别的情况[1]。我们应该从计算原因而非提高准确度的角度考虑它。

多标签分类模型。如果希望每个输出有多个分类（例如“人 + 车”），不应将 softmax 作为输出层激活函数，而应在有多个输出单元的输出层使用 sigmoid 激活函数，每个输出单元代表不同的类别，然后每个单元分别给出每个类别的概率。

在这种场景中，再次对输出层使用二元交叉熵（`LossFunction.XENT`）损失函数。

训练数据时我们将输出向量的数据表示成每个类别为 0 或 1。这与前面提到的独热表示不同，因为有多个标签条目的值可以为 1.0，而独热只能有一个 1。



多个输出层

假设用神经网络预测汽车制造商（福特、丰田、通用等）以及车的类型（SUV、跑车、卡车等）。在这种情况下，需要使用 ComputationGraph+ 多个输出层。欲获取更多信息，请访问 https://deeplearning4j.org/compgraph#multitask。



快速参考：softmax 与 sigmoid 激活函数对比

softmax 激活函数将输出的和限制为 1.0（即概率），而 sigmoid 激活函数则分别限制输出值。例如 softmax 层输出的概率全加起来才到 1.0，而 sigmoid 层每个单元的输出值都有可能是 0.9，sigmoid 层没有这种“横向”约束。

6.4 处理层的数量、参数的数量和存储器

在神经网络中，层的数量和每层神经元的数量决定模型参数总数量。网络中参数总数量取决于层的类型以及每层神经元之间的连接（或权重）的数量（加上偏置权重）。



层的类型与神经元数量

所使用的层的类型也会影响网络中参数的数量。对于同样的层输入 / 输出大小，LSTM 层的神经元比普通循环神经网络层更多，在 DL4J 中循环神经网络有多个 `DenseLayer`，而 `DenseLayer` 又比大小与之相似的 CNN 层有更多参数。

随着参数数量增长，需要建模的功能变得更加复杂。超过某个点之后，参数的数量多到创建了一个过拟合了数据集中太多细微特征的模型，这样的模型不能泛化。

注 1：分级 softmax 不是唯一的解决方案，关于另一个解决该问题的替代方案，请参阅 Vishwanathan 等人于 2015 年发表的文章：“BlackOut: Speeding up Recurrent Neural Network Language Models With Very Large Vocabularies”。

对于不同的网络架构，设置层、神经元和连接权重的方式存在着变化。有一些针对神经网络的高层次思路主要应用于前馈多层感知器神经网络，本章稍后结合具体架构详细介绍。

6.4.1　前馈多层神经网络

对于前馈多层感知器神经网络，输入层的输入单元需要与输入向量的数量相同，输出层将简单地等于需要分类的标签数量和用于回归的单个神经元。接下来将讨论确定神经网络的层和神经元数量的策略。

1. 决定隐藏层的数量

隐藏层的数量和数据集大小也有关系。随着数据集大小的增长，也需要更多隐藏层。例如MNIST，它只有大约 3 到 4 个隐藏层（超过这个深度后，准确度会降低），但是 Facebook 的 DeepFace 使用了 9 个隐藏层，而我们猜测那是一个非常大的数据集。



隐藏层数量的经验法则

隐藏层的经验法则是数据集越大，需要的隐藏层和神经元就越多，而不会带来过拟合。因为有更多的训练数据，就不太可能过拟合。一个更大的网络可能达到更高的准确度。如果网络规模太小，并且输入数据集较大，会陷入欠拟合的风险，并且数据集的准确度不会达到最佳。

对层和参数数量影响更大的另一个因素是数据集的较大变化。

2. 决定每层神经元的数量

如果网络在隐藏层中的神经元太少，将很难在训练时很好地对数据建模。如果网络中有太多的参数，我们要么需要处理过拟合问题，要么耗费更多的处理时间来找到一个拟合良好的模型。

考虑神经元数量的一个很好的思路是：层的神经元数量应该逐渐下降。也要注意隐藏层中节点数量不少于输入层中节点数量四分之一的情况。如果隐藏层中神经元过多，也会增加数据集过拟合的可能性。



层中神经元数量的经验法则

层大小、正则化和数据量需要被正确组合。大的层加上正则化不足会不利于泛化。更大（和更多）的层（技术上来说，意味着更多参数）通常需要更积极的正则化（或更多数据）以避免过拟合。

有些情况下，对所有层使用相同数量的隐藏参数的做法比减少神经元数量的做法效果更好，然而这种效果可能只在某些类型的数据集上起作用。

6.4.2　控制层和参数的数量

通过去除神经网络中的参数（或神经元）来解决诸如过拟合之类的训练问题是一种常见的（一开始就能想到的）思想，而实践中往往设置隐藏层神经元数量超过最优数。

经验表明，当使用无监督预训练（DBN 中的 RBM）神经网络时，隐藏层神经元的最优数要大得多。一个实际的网络，隐藏层神经元数量可能在数百个单元到数千个单元之间。而为了防止过拟合，建议使用以下正则化技术：

- L1
- L2
- Dropout
- 输入噪声
- DropConnect

原因是较小的网络的局部极小值将很少（但可能是坏的模型），而较大的网络的局部极小值将更多。不使用较小的网络，因为担心会过拟合训练数据集。相反，只要计算平台允许，就可以向问题抛出尽可能多的参数，并使用上述技术来克服过拟合。



不要过分依赖参数

虽然前面说在这个问题上可以抛出更多的参数，但要实事求是。单单一层的神经元就达到 100 万的做法可能不是最好的，即使硬件有能力处理它。

随着训练数据集的变化和大小的增加，建议慢慢增加隐藏层数量和每层的神经元数量。



添加更多数据以抑制过拟合

抑制过拟合的最佳方法之一是（在可能的情况下）使用更多的训练数据。

确定网络的参数数量

在实践中有两种方法确定网络的参数数量。

- 手动计算某种架构的参数数量。
- 使用 DL4J API 计算参数数量。

可以简单地把每层的参数数量相加，计算出网络中的参数数量。表 6-2 给出了一些常用层的参数数量。

表6-2：计算每种网络层的参数数量

层类型	参数数量
全连接（如 DL4J 的密集层、MLP、输出层）	$n^{L-1}n^L + n^L$
卷积	$d^{L-1}d^L k^H k^W + d^L$
LSTM RNN（如 GravesLSTM）	$4n^L(n^{L-1}+1)+n^L(4n^L+3)$
标准（普通）RNN	$n^{L-1}n^L+(n^L)^2+n^L$

以下是参数计算的说明。

n^{L-1}

输入的数量（前一层 $L-1$ 的大小）。

n^L

当前层 L 的大小。

k^H 和 k^W

卷积层的核的高度和宽度。

d^{L-1} 和 d^L

卷积层的输入和输出深度 / 通道。

注意，一些类型的层没有任何参数，例如子采样、激活层以及 DL4J 的 `LossLayer`。一个复杂的情况是网络混有卷积层和密集（或输出）层。对于密集（或输出）层，在卷积 / 子采样层之后，需要确定最后一个卷积层的激活数组的大小。我们把它的长度（每个样本）作为密集 / 输出层的 n^{L-1} 值。例如，如果最后的卷积层输出了 100 个通道的 5×5 激活值，那么应该使用 $n^{L-1}=5\times5\times100=2500$ 作为密集 / 输出层的输入[2]。

假设有一个 DL4J 配置，那么该如何检查参数的数量呢？这里有一个简单的方法：

```
MultiLayerConfiguration configuration = ...
MultiLayerNetwork network = new MultiLayerNetwork(configuration);
network.init();

System.out.println("Total number of parameters: " + network.numParams());
for( int i=0; i<network.getnLayers(); i++ ){
    System.out.println("Layer " + i + " number of parameters: " +
        network.getLayer(i).numParams());
}
```

6.4.3 估计网络内存需求

如果网络配置超过了运行时使用的硬件的能力，那么当内存完全耗尽时，训练将失败。虽然严格来说这与调优无关，但是对大型网络，这很重要——毕竟如果不能先训练网络，就谈不上调优网络。

当训练神经网络时，内存主要用于分配多维数组（DL4J 中的 `INDArrays`）。在训练神经网络时，这些数组有六种不同的用途：

- 网络参数
- 参数梯度（与网络参数大小相同的）
- 网络激活值
- 网络激活值梯度（与网络激活值大小相同）
- 更新器状态（即动量或 `RMSProp` 的历史值，参数数量的整数倍）
- 训练数据

此外需要考虑工作内存 / 临时数组、形状缓冲区信息、异步预加载数据以及其他所以需要 JVM 运行的事务的开销。

注 2：有关计算输入大小的更多细节，请参阅斯坦福大学的 CS231n 课程笔记和 DL4J 的 `ConvolutionMode` 的 JavaDoc。（注意，也可以使用 DL4J 的 `InputType` 功能计算和设置 `numInputs` 值，并使用 DL4J 配置网络。）

数学上，训练神经网络（以字节为单位）的最小内存需求可以根据以下表达式估计。

$$N_{\text{bytes,test}} = 4[\,n_{\text{params}}\,(2+u)+m(2a+d)]$$

而在测试阶段（训练后）使用网络，我们只需要：

$$N_{\text{bytes,test}} = 4\,[\,n_{\text{params}}\ +m(a+d)]$$

其中：

- u 是更新器的大小（$u = 0$ 为 SGD，$u = 1$ 为动量、RMSProp 和 Adagrad；$u = 2$ 为 Adam 和 Adadelta 更新器）；
- m 是小批量大小；
- a 是单个样本（所有层）的网络激活值的大小；
- d 是单个样本的大小。



并行处理与内存计算

当训练扩展到多台机器时（使用 Spark 或 `ParallelWrapper`），需要乘以 $N_{\text{bytes, test}}$ 和模型副本的数量。

关于 DL4J、内存和精度的说明

默认情况下，DL4J（通过 ND4J）对所有 `INDArray` 使用 32 位浮点（FP32）数字，这意味着一个带有 N 个元素的 `INDArray` 将消耗 $4N$ 个字节。也可以将 64 位浮点数字（FP64）用于网络训练，有时它有助于解决数值稳定性问题（尽管数值稳定性问题通常表示有其他调优问题存在），然而 FP64 更消耗内存（2 倍）和性能损失（在普通的 GPU 上最多为 32 倍，在 Tesla GPU 上最多为 2 倍）。由于对内存和缓存的影响，使用 FP64 时 CPU 性能也将受到较小程度的影响。

最后还需注意，16 位浮点（FP16）格式也可以在 DL4J 中用于在 GPU 的 CUDA 上进行训练。与默认的 FP32 数据格式相比，FP16 格式的内存需求将减少一半，从而支持更大的网络、更大的批量或两者兼有，然而 FP16 内存的节省是以显著降低数值精度为代价，这会使得网络更难调优。除非真的需要，否则如果刚开始调优神经网络，或者在调优网络时遇到困难，请使用默认的 FP32。至于表现，FP16 在大多数普通的 GPU 上表现非常差，因此在 DL4J 中 FP16 仅用作存储格式。实际操作在首先将数据转换为 FP32 之后执行。

如果遇到给定硬件的内存问题，有以下几种选择：

(1) 降低小批量大小 m；
(2) 使用更好的硬件（或使用云服务，如 Azure、亚马逊云服务或谷歌云）；
(3) 在 GPU 上使用 FP16 代替 FP32（但要注意数值精度的降低）；
(4) 使用较小的网络。

6.5 权重初始化策略

权重初始化会极大地影响训练过程，应该注意使用适当的权重初始化策略，使其与合适的问题场景相匹配。初始化权重是神经网络和深度网络学习时的一个关键起点。

通常使用以下策略作为权重初始化的基本策略。

- 偏置通常初始化为 0。
- 隐藏层的权重需要初始化，这样它们就打破了隐藏单元之间的对称性。

通过在权重初始化过程中加入随机性来打破隐藏单元之间的对称性。权重初始化应该是输入数量（也可能是输出）的函数。



在使用 ReLU 或 leaky ReLU 激活函数时，使用 `WeightInit.XAVIER` 或者 `WeightInit.RELU` 初始化权重。

如果权重太大，则意味着输出值大，梯度值大，这在学习过程中显然会产生有害的影响。初始化网络单元的偏置也需要处理。学习的早期阶段会受到初始化可见单元偏置方式的影响。初始化网络偏置有不同的方法，但通常默认初始化为 0。0 的初始隐藏偏置通常在特定的稀疏概率的场景之外表现都很好。

本节介绍的两个具体情况分别是连接到 tanh 和 ReLU 单元的权重。通常它们是通过从有典型方差的（通常是高斯或均匀）分布采样来随机设置的。在很多情况下，可以基于以下指导方针采取行动。

- 对于 ReLU 家族激活函数（ReLU、leaky ReLU、随机 leaky ReLU 等），在 DL4J 中使用 `WeightInit.RELU` 来初始化权重。
  这种方法基于 He 等人于 2015 年发表的论文“Delving Deep into Rectifiers”，在某些情况下也被称为“He 初始化”。
- 对于其他大多数激活函数（tanh、恒等函数等），使用 `WeightInit.XAVIER`。
  这种方法基于 Glorot 和 Bengio 于 2010 年发表的论文“Understanding the Difficulty of Training Deep Feedforword Neural Networks”，也称“Glorot 初始化”。
  对于连接到 tanh 单元的权重，其他的初始化方法有稀疏初始化等。
- 偏置通常被初始化为 0（DL4J 中的默认值）。

简而言之，ReLU 和 Xavier 的初始化基于网络架构（层大小）和对激活函数的假设来设计，它们做以下两件事情。

- 确保网络激活的变化对于所有层都是恒定的。例如，在网络后面的层中，激活值不会变得太大或太小。
- 确保所有层的激活值（和参数）的梯度变化是恒定的。例如，梯度在反向传播到前面的网络层时不会变得太大或太小。



权重初始化和满足两个条件

ReLU/Xavier 初始化不能在层大小不同（或者如果激活函数假设不成立）的网络中同时优化这两个约束（激活 / 梯度的变化）。

RNN的正交权重初始化

第 7 章将研究特定架构（如 LSTM）的调优策略。值得注意的是，LSTM 和门控循环单元（GRU）在一些具有正交权重初始化的情况下表现更好。

6.6 使用激活函数

表 6-3 是对前馈网络中与数据分布类型相匹配的一些激活函数的说明。

表6-3：前馈网络中目标分布与输出层激活函数的关系

目标分布	输出层激活函数
二元分布（0, 1）	sigmoid 激活函数
分类目标（1-of-*K* 编码）	softmax 激活函数
连续值（有界范围）	sigmoid 或 tanh（将输出范围缩放至目标范围）
正值（没有已知的上界）	ReLU 家族激活函数、softplus 激活函数（或使用对数规范化转换为无界连续值）
连续值（无界）	线性激活函数（等同于无激活函数）

在神经网络文献中，sigmoid 函数最为常见。在二十年前的神经网络实践中，sigmoid 类的函数很流行，今天它们很大程度上已经不再适用于隐藏层了。



sigmoid 与信息丢失

与 ReLU 激活函数相比，由于前向和反向传播中的饱和，sigmoid 函数更容易丢弃信息，在小的邻域内由于单个参数而给网络带来非线性影响。

sigmoid 存在大量输入零梯度问题。这对小批量处理来说不是一个问题，但是需要注意初始化权重以避免饱和。

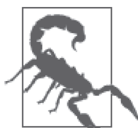


sigmoid 激活功能的下降

人们现在更青睐 leaky ReLU 激活函数而非 sigmoid 激活函数。leaky ReLU 激活函数既没有 sigmoid/tanh 的梯度消失问题，也没有普通 ReLU 的“死亡 ReLU”问题。

最好不使用 sigmoid 隐藏层。

ReLU 激活函数（分段线性单元）是如今现代深度网络中最流行的隐藏单元类型，在 CNN 中 ReLU 常用作卷积层的激活函数。人们还发现，与 sigmoid 和 tanh 函数相比，SGD 能加快 ReLU 的学习。ReLU 在计算上的成本也比 sigmoid 和 tanh 函数更低。



死亡 ReLU 问题

在使用 ReLU 时必须小心，因为有些单元可能永远不会在整个训练数据集上被激活（有时称为“死亡 ReLU 问题”）。

网络仍然可以在包含许多“死亡”ReLU 单元的情况下学习，然而这些单元不会对网络输出做出贡献，导致了算力的浪费及网络有效容量变低。

最好的做法是使用 leaky ReLU 变体，它（与普通的 ReLU 不同）对于所有输入值都具有非零梯度。

hard tanh 激活函数存在与 ReLU 激活函数类似的零梯度问题，它也在部分输入范围上为零梯度。具体说来，对于小于 −1 和大于 +1 的输入，梯度为零。

关于 Maxout 的说明

Maxout 模型是使用 Maxout 单元的前馈网络。Maxout 单元被视作 ReLU 的推广，但没有死亡单元的问题。Maxout 单元也使得每个神经元的参数数量变为两倍，增加了总体的参数数量。

Maxout 单元能够在 ReLU 不能很好工作的场景中学习。ReLU 和 Maxout 单元的区别在于 Maxout 单元是分段线性的，其中每一段线性函数都有自己的权重向量。Maxout 单元的这个特点使得它避免了 ReLU 单元容易出现的在某些地方学习停滞的问题。

激活函数汇总表

表 6-4 总结了本节所讨论的激活函数。

表6-4：常用激活函数及其使用方法列表

函数名	使用场景
线性	回归的输出层
sigmoid	• 二元分类输出层 • 输出值范围在 [0, 1] • 失去青睐，不要在隐藏层使用
tanh	• 连续数据，范围在 [−1, 1] • LSTM 层
softmax	多类别分类模型的输出层
ReLU	• RBM • CNN 层 • 多层感知器网络层

通常建议尝试 ReLU 激活函数以供一般使用，但要注意学习率以及网络中的死亡神经元。为了克服这些影响，可以尝试 leaky ReLU 激活函数或 Maxout 激活函数。tanh 激活函数是另一个替代函数，但实践中发现它执行起来比 ReLU 和 Maxout 激活函数差。sigmoid 单元通常仅用于输出层的单标签分类，并且已经不再适用于隐藏层。

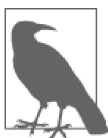
6.7 应用损失函数

损失函数让优化函数知道它在预期任务中表现如何。在前面的章节中，我们参考了损失函数在模型目标（如输出层损失函数）场景中的使用。以下是关于损失函数在其他场景的一些说明：在哪里使用它们最高效，以及使用时的注意事项。简单说来，损失函数只在两个地方使用。

- 输出层（或损失层）。
- 支持无监督逐层预训练的层，例如自动编码器和 VAE。

大多数层（卷积层、密集层、LSTM 等）没有（也不能有）损失函数，因为它们不能以逐层的方式预处理。

除了本章前面列出的分类损失函数外，其他分类损失函数包括 hinge 损失和逻辑损失。当网络必须针对硬分类进行优化时，hinge 损失是最常用的损失函数。例如 0= 非欺诈和 1= 欺诈，它习惯上被称为“0-1 分类器”。当我们对概率比对硬分类更感兴趣时，使用逻辑损失函数。例如使用人工循环解决方案标记潜在的欺诈，或者在点击广告可以得到 1 美元收益的前提下，预测某人点击广告的可能性。预测有效概率意味着生成一个 0 到 1 之间的数字。



二元分类与 hinge 损失

我们主要使用 hinge 损失来进行二元分类。hinge 损失有对多类别分类的扩展（如一对所有，或一对一），这里没有涉及。

理解训练过程中的调试输出

在训练期间，将看到如下所示的命令行输出：

```
21:36:00.358 [main] INFO  o.d.o.l.ScoreIterationListener
  -  Score at iteration 0
    is 0.5154157920151949
```

最后的值是当前小批量中每个样本的损失函数的平均值。这个数字并不总是以相同级别的数值开始和结束，并且取决于网络架构使用的损失函数。比如说，使用 MSE 与使用负对数似然损失函数相比，会得到不同的进度分数。

这里有个最简单的方法来理解这个数字：“低为好，高为坏”，通常希望它随着时间下降。

若要打开调试输出，请将以下代码添加到示例中：

```
myNetwork.setListeners(new ScoreIterationListener(1));
```

表 6-5 总结了每个损失函数的使用场景。

表6-5：损失函数及其使用场景总结

损失函数	使用场景	属性
重建熵	RBM、自动编码器	用于特征工程
平方损失	输出层	回归
交叉熵	输出层	二元分类
多类别交叉熵	输出层	分类
均方根误差	自动编码器、RBM、输出层	特征工程、回归
hinge 损失	输出层	分类
负对数似然	输出层	分类



交叉熵、逻辑损失与负对数似然

文献和示例中可能以类似的方式使用这些损失函数。如第 2 章所述，交叉熵起源于信息论，而负对数似然分类起源于统计建模。这些方法在数学上相同，所以尽管无须在意使用的是哪个，但它们往往会令人费解。

6.8 理解学习率

神经网络中的学习率超参数是对神经网络调优时要设置的最重要的超参数之一（或者去掉“之一”）。它对训练的稳定性和时间效率都有很大的影响。

从图 6-1 可以看出，如果学习率过高（左图），网络训练可能不稳定（或完全发散）。如果学习率过低（右图），可能比正常训练耗时更多。

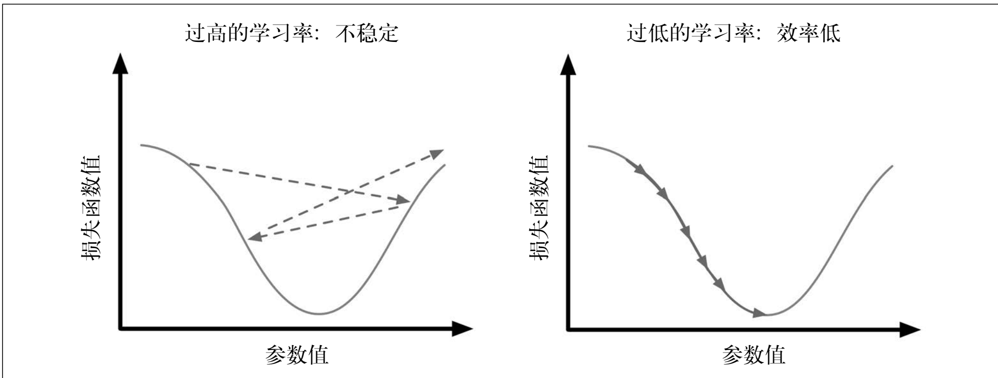


图 6-1：SGD 与学习率

理想情况下，在训练开始时使用一个高的学习率，接近训练过程的收敛时，让它随时间而下降。如果正在使用动量法，那么随着学习率的下降，在训练时慢慢增加动量值（稍后将介绍）是有利的。



动量和某些更新器

如果使用的是某些更新器，如 Adam 或 RMSProp，那么动量不会被使用。



增大学习率设置值

增大学习率的初始设置值并不总是有用的，因为尽管最初可能学习得很好，但从长远来看，算法反而学习得更慢。

理想情况下，使用某种非静态学习率方案，其中学习率可以设置在多层：

- 全局
- 每层
- 每个神经元
- 每个参数（其中每个神经元有更多的参数、偏置等）

学习率应该只在学习接近尾声时才开始下降。安排学习率随时间衰减很必要，这就好比可以利用额外计算时间使训练时间更长，以降低衰减率一样。这个衰减率应适应每一个参数，而不是过于激进、过快地妨碍学习。

6.8.1 使用参数更新比率

设置学习率的一种简单高效的方法是使用参数更新比率（或者具体说来就是它的平均更新幅度）。回想一下，更新是将更新器（RMSProp、动量等，以及学习率）应用于梯度后的值，即学习方程应用了更新 $\boldsymbol{u}$ 后变为 $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \boldsymbol{u}$。如果令更新向量为 $\boldsymbol{u}$（长度为 N，与参数向量长度相同）、参数向量为 $\boldsymbol{\theta}$，那么参数更新比率的计算如下所示：

$$\text{参数更新比率} = \frac{\frac{1}{N}\sum_{n=1}^{N} |u_i|}{\frac{1}{N}\sum_{n=1}^{N} |\theta_i|}$$

这个比率测量了相对于当前的参数值，参数的值改变了多少。可以在 DL4J 的 UI 中的两个地方找到这个比率：Overview 页面（左下方）和 Model 页面（在选择层的顶点之后的第一个图）。请注意，在 DL4J 的调优 UI 中，这个比率以 10 为底的对数值表示，如图 6-2 所示。

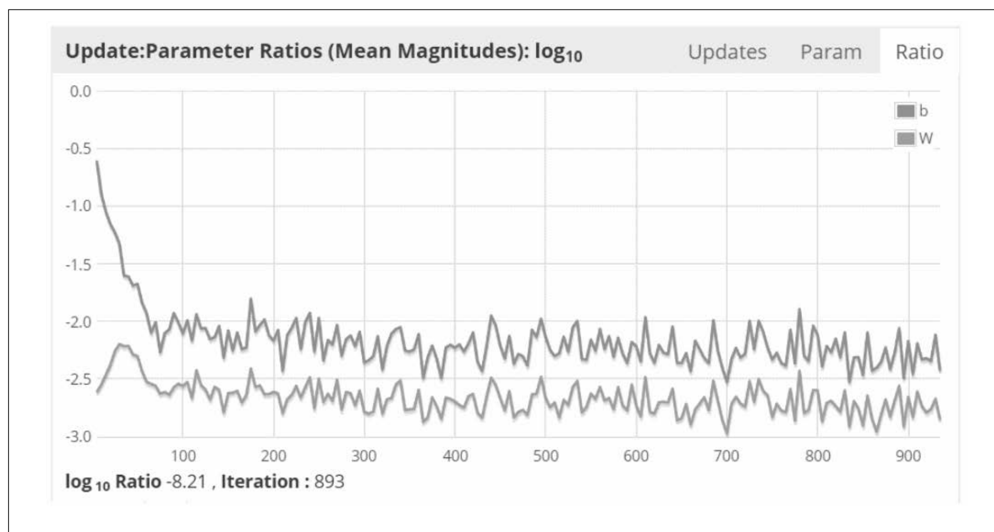


图 6-2：DL4J 的调优 UI 显示了参数更新比率图

当使用 DL4J 时，为了查看图 6-2 中所示的训练进度，在网络开始训练之后，在浏览器中访问 http://localhost:9000/。



利用比率设置学习率

为了设置学习率，需要一个大约 0.001 的参数更新比率，这与 DL4J 的训练 UI 中的 $\log_{10}$ 比率等于 −3 的值相对应。

例如对数比率 −4（即 0.0001）意味着应该提高学习率，对数比率 −2（即 0.01）意味着应该降低学习率[3]。

6.8.2 关于学习率的具体建议

学习率设置往往从较大的值开始，并通过衰减机制随时间下降。如果学习率初始值太大，通常会导致平均训练损失增加。最好的学习率通常接近（两倍于）不会导致学习过程发散的最大学习率。



良好的初始学习率

理想情况下，从高的学习率开始，如果学习过程发散，就将这个速率除以一个因子，然后尝试，直到学习过程开始收敛。

建议从这几个学习率开始：[0.1, 0.01, 0.001]，看看哪个最稳定，其中 0.001 是最受欢迎的学习率初始值之一。

注 3：为实现这个目标，还可以使用其他度量，例如 L2 范数的比率。这些在 DL4J 的训练 UI 中不可用，但是可以手动计算它们。

建议按以下顺序尝试学习率方法。

(1) Adam：
    目前推荐使用的默认方法。

(2) Nesterov 动量。

(3) RMSProp：
    a. 使用平方梯度的移动平均；
    b. 当学习率变得单调降低时没有问题，就像在 AdaGrad 中一样。

(4) AdaGrad。

DL4J 支持的其他选项包括：

- AdaDelta



仍然需要观察学习率

如果学习率设定得太高，用 AdaGrad 训练仍然是不稳定的（就像其他方法一样）。AdaGrad 不能提高学习率，实际上使用 AdaGrad 时的有效（每个参数的）学习率是单调下降的。

AdaGrad 确实表现得像是每个权重学习率的衰减机制，然而 AdaGrad 的缺点是它单调的学习率变化机制通常在实践中表现得过于激进，并且过早地停止学习。

DL4J 的库内置了 AdaGrad 的实现，因此用户可以不用太操心手动调整学习率。



在 DL4J 中使用动量法或 Nesterov 动量法

动量法能够避免学习过程陷入局部极小值，并能为优化过程找到更好的解决方案。

动量值通常都在 0.9 左右。关于这个设置的其他常用值有：[0.5, 0.9, 0.95, 0.99]。动量的典型初始设定值是 0.5，经历多轮训练后变为 0.9 左右。除了设置 `.momentum(0.5).` 之外，设置 `.updater(Updater.NESTEROVS)` 也是至关重要的。如果没有这两个设置，动量就不会被启用。

当隐藏单元的输入（其中有许多连接进入）非常大时，需要设置较小的更新，因为同一方向上很多小的改变可能对梯度符号产生意想不到的影响。在这种情况下，学习率应该更低。偏置的更新可以较大，目前还未看到不利影响。这些机制使学习率的调整成为一个挑战。

这里还有一些其他的事情要牢记。

- 学习率是特定于网络的：在一个网络中工作正常的值可能在另一个网络中不起作用。
- 请记住，参数更新启发式方法仅仅是一个指南，它并不总是有效的，但它通常是一个好的起点。
- 所需的学习率（有时尤其）取决于其他超参数。例如不同的更新器（动量法、RMSProp、AdaGrad 等）可能需要不同的学习率。改变权重初始化或层的数量 / 类型 / 大小也可能需要重新调整学习率。因此，对网络进行重大调整之后，应该经常检查学习率。

- 在阅读更新比率图表之前，给网络一些时间来稳定下来——也许训练开始后要迭代几十次到几百次（参数更新）。训练开始后的最初几次更新通常比较大。
- 偏置更新率的初始值比 0.001 大很多是正常的，也是符合预期的，毕竟偏置通常被初始化为 0.0。因此，初始更新率将会很大，并会随着时间推移减小幅度。
- 在某些情况下，为了获得正确的比率，有必要为不同的层设置不同的学习率。然而这也可能引发其他问题，例如梯度消失 / 梯度爆炸、较差的权重初始值或数据规范化问题。Adam 和 RMSProp 更新器可能有助于缓和这些问题，因为它们在梯度上具有规范化 / 缩放效应。

6.9 稀疏性对学习的影响

稀疏超参数是深度学习中常用的参数。稀疏性鼓励隐藏单元稀疏（或接近零）。稀疏表示被视作好的，因为它们鼓励底层因子的分解。

稀疏性有助于在权重值很大以及有问题的时候，训练不会停止。稀疏性有时与决定网络规模的其他超参数相互作用。随着隐藏节点数量的增加，稀疏因子应该增加。稀疏性与网络大小的相互作用也会受到网络架构所使用的激活函数类型的影响。

为了理解稀疏性设置如何工作，请查看隐藏单元平均活动的直方图。由此可以设置稀疏性，使隐藏单元的平均概率在目标附近。如果概率紧密聚集在目标值周围，需要减少稀疏设置，这样它对学习过程主要目标的影响较小。



稀疏性与某些网络架构

对于 DL4J 中的 `DenseLayer` 和 LSTM 层，稀疏性不可用。



稀疏目标值

为了有质量地学习，将稀疏目标值设置在 0.01 和 0.1^{-9} 之间。

6.10 优化方法的应用

最流行的优化深度学习网络的方法是 SGD，它相对容易实现，但在调优和并行化上有挑战性。

一些研究人员认为，给神经网络增加额外的参数并不能减少大数据集上的欠拟合和容量的浪费。他们指出这是 SGD 的原因，并建议使用二阶方法缓解这些问题，这些方法包括：

- Broyden-Fletcher-Goldfarb-Shanno（BFG）——有限记忆 BFGS（L-BFGS）
- 共轭梯度（CG）
- Hessian-free

表 6-6 列出了对于每个网络架构，初始优化方法最佳实践的快速参考表。

表6-6：不同网络架构的优化算法

网络	常用的训练方法及说明
DBN	SGD
CNN	SGD (+Dropout)
RNN	SGD, Hessian-free

需要注意训练数据集大小对优化方法选择的影响。当处理较小的数据集时，建议使用二阶优化方法，并将批量大小设置为整个数据集的大小。当处理较大的数据集时，建议使用带有小批量设置的 SGD。你也可以尝试二阶方法，并将小批量大小设置为小于整个数据集的大小。

表 6-7 给出了每种方法的比较。

表6-7：优化方法比较

方法	一阶信息	二阶信息	优点	缺点
SGD	梯度	无	收敛速度快，每个参数的更新成本最低	不够稳健
L-BFGS	梯度	由梯度估计的曲率信息	能够找到更好的局部极小值	能够比 SGD 训练得更好，但每个参数更新成本更高，内存占用更高
CG	梯度	由梯度估计的曲率信息	—	每个参数更新成本更高，内存占用更高
SGD Hessian-free	无	曲率	自动推导出步长，使用共轭梯度找到下一步	不能泛化到所有架构，每个参数更新成本更高，内存占用更高

尽管提供了表 6-7 用于快速比较，但是大部分情况下你应该关注如何从 SGD 开始，因此稍后将深入探讨它的具体最佳实践。

SGD最佳实践

SGD 与动量和精心设计的随机权重初始化方案工作良好。它已被证明在训练 DBN 和 RNN 上非常高效，达到了与 Hessian-free 优化方法相同的水准。研究人员发现，在大型分类问题上很难找到比精心调优过的 SGD 方法更好的方法。

下面是在深度网络中使用 SGD 训练的快速入门总结。

- 在输入训练集上使用好的数据混洗方法。
- 观察每次迭代中的误差百分比和验证误差率：
  - 训练误差变小；
  - 如果验证误差变平，就可以提前停止训练。
- 使用训练数据的较小子集来尝试调整超参数（特别是用于调整学习率）。
- 与动量、AdaGrad 和 RMSProp 结合使用。
- 适当设置学习率，太高或太低都不好。
- 将输入数据规范化（不只限于 SGD，但值得反复提醒，因为它经常被忘记）。

对于实值输出的较小问题（函数逼近、控制问题等），CG 方法的表现很好。使用二阶方法时的小批量大小（10 000）通常比使用 SGD 方法时更大。



将 SGD 作为最常用的起点

在实践中，比起二阶方法，人们更多地使用 SGD（结合使用动量或 AdaGrad 或 RMSProp 等）。

6.11 使用并行化和GPU更快地进行训练

随着机器学习应用目标变得更加雄心勃勃，需要更大的模型来配合它。这意味着需要训练更多的参数和训练时间。训练更大的数据集可以获得更完整的模型视图，但是当数据达到一定规模后，会看到 I/O 开销碰到了天花板（参考附录 C 中 Jeff Dean 的 12 个数字）。在单机训练的情况下，受限于现代 CPU 的原始时钟频率，计算机架构已经达到硬件能力的极限，人们将多个 CPU 内核的计算能力组合在一起，以与之前不同的并行处理的方式，获得了新的增长。在现代机器学习活动中，很快就达到了顺序学习的极限。单机的存储和网络带宽已经跟不上数据量的增加，我们需要考虑能够以分布式方式执行数据分析算法的步骤。



访问磁盘上的大量数据

访问硬盘驱动器是耗时的操作，会导致计算迟延。例如，假设一个容量为 2TB 的硬盘驱动器，磁盘读取速度为 100MB/s，那么读取全部内容需要耗费大约 6 个小时。有关计算机架构不同迟延的更多信息，请参考附录 C 中 Jeff Dean 的 12 个数字。

6.11.1 在线学习与并行迭代算法

从批量学习算法到在线学习算法的转变至今已有十多年了，转变后模型能够处理不断增加的数据集大小。然而当有多个数据驱动器时，这个任务在计算上变得不切实际：I/O 开销导致无法通过 SGD 处理所有数据。在大数据的世界中，需要尽量少地移动数据，所以我们最终寻找并行执行程序的方法。

在最近的工作中，传统的串行学习过程已被重新设计用于并行计算，即通过让一个内核执行本地计算，然后将多个内核的工作集中组合以产生全局结果。随着数据集超过多个驱动器的容量，我们需要重新考虑设计存储和处理系统。这是 21 世纪初谷歌的 MapReduce 和谷歌文件系统设计的推动力（以及之后 Doug Cutting 和 Michael Cafarella 继续在 Hadoop 上完成的工作）。

可以用不同的方式并行化计算机程序，具体说来就是学习算法。任何并行程序都由同时执行的进程组成，并行性的不同形式与如何将进程分解成可以在不同位置并行运行的部分进程有关。实现它的两个主要的方法是**任务并行性**和**数据并行性**。

1. 任务并行性

利用任务并行性（也叫**函数并行性**），将要完成的工作划分为几个独立的任务，然后将这些任务调度在不同的处理单元上执行。处理单元可以是本地线程，也可以是不同物理机器的内核。

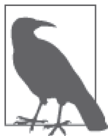
2. 数据并行性

数据并行性的意思是划分工作，将同样的函数应用于数据集的不同部分，然后执行工作的任务在不同的线程（本地或分布式集群的远程机器中）上被调度，这使得数据并行性成为任务并行性的子集。通过数据并行性，我们在并行计算环境中可以跨多个处理单元扩展计算。这些单元可以是同一内核上的线程、不同内核上的线程，或是集群中完全不同的物理机器的内核。

当并行化执行 SGD 之类的迭代类算法时，可以使用参数平均技术来高效地利用本地线程并行性。然而随着数据大小的增加，把数据复制到并行处理单元就会成为一个问题。为了解决这个问题，第一个思路就是迁移到一个能够处理大数据的系统，比如 Hadoop。

大数据

在企业软件的市场营销中，经常能听到“大数据”这个词。这个术语已经变得无处不在，甚至被总统提及，并且经常在《华尔街日报》中被谈论。非常有趣的是，尽管它被如此广泛的使用，大多数人依然很难定义这个词。我们基于实践经验这样定义大数据：数据在存储的地方被处理，我们将计算移动到数据。MapReduce 就是围绕这一概念构建的。许多公司和工具声称要处理大数据，但简单说来，如果在处理数据之前移动数据，那么处理的就不是大数据。



扫描一 PB 数据需要多久？

大数据的一个限制是在达到某个点之后数据的移动变得不可能完成。例如计算以单个普通硬盘驱动器的速度（大约 40MB/s）线性扫描一 PB 数据需要多久。如果这种基本的扫描 / 复制操作以 40MB/s 的速度在单个处理单元上进行，将需要大约 310 天。

Apache Hadoop 是存储和管理大量数据，（如 Web 日志或传感器读数）的好地方，因为在世界各地的企业中已经证明了其对 PB 级数据的处理能力。最早包含在 Hadoop 发行版中的传统并行框架是 MapReduce 框架。MapReduce 并行化利用了存储大量数据的位置，它将计算（或任务）推送到存储数据块的本地主机，并在那里本地执行以最大化物理存储上的吞吐量。

MapReduce

MapReduce 是受到 map 和 reduce 函数启发的传统批处理类并行化算法，通常用于函数式编程。它处理来自 map 任务的键值对（在具有数据块的主机上本地执行，以减少 I/O 和网络开销），将数据混洗到分区，以便每个 reducer 在归约阶段操作。诸如 Hadoop 处理机器故障之类的实现对程序员来说是透明的，而传统上这是并行编程的一部分。然而由于数据集之间数据遍历的时间成本问题，MapReduce 不适合迭代类算法。



迭代算法与 MapReduce 问题

如果要遍历数据集 100 次，并且 MapReduce 调度成本是每次遍历 30 秒，那么这会带来 3000 秒（100 × 30）的开销。这还仅仅是调度开销时间的 50 分钟，最终将影响许多训练过程。这种开销使 MapReduce 对于并行迭代算法来说不是一个良好的并行框架。

近年来，并行优化方法作为机器学习算法的一种扩展方式越来越受到人们的关注。从 MapReduce 方式的并行到更多迭代并行方法，各种形式的并行在加速。SGD 一直是许多研究论文的重点，这些论文在没有使用 MapReduce 框架的情况下已经并行化了 SGD。开源框架（Vowpal Wabbit 和 Apache Spark）在迭代方法的并行化方面也取得了进展，参数平均是迭代方法实现的关键。

6.11.2 DL4J中的SGD并行

谷歌发布的另一篇论文描述了如并行 SGD 这样的用于迭代类算法的并行系统。这个系统被称为 Downpour SGD，它利用了某些特性，如 AdaGrad、大量模型副本以及 Sandblaster（并行 L-BFGS）。Downpour（及其组件）启发了 DL4J 并行化架构，如图 6-3 所示。

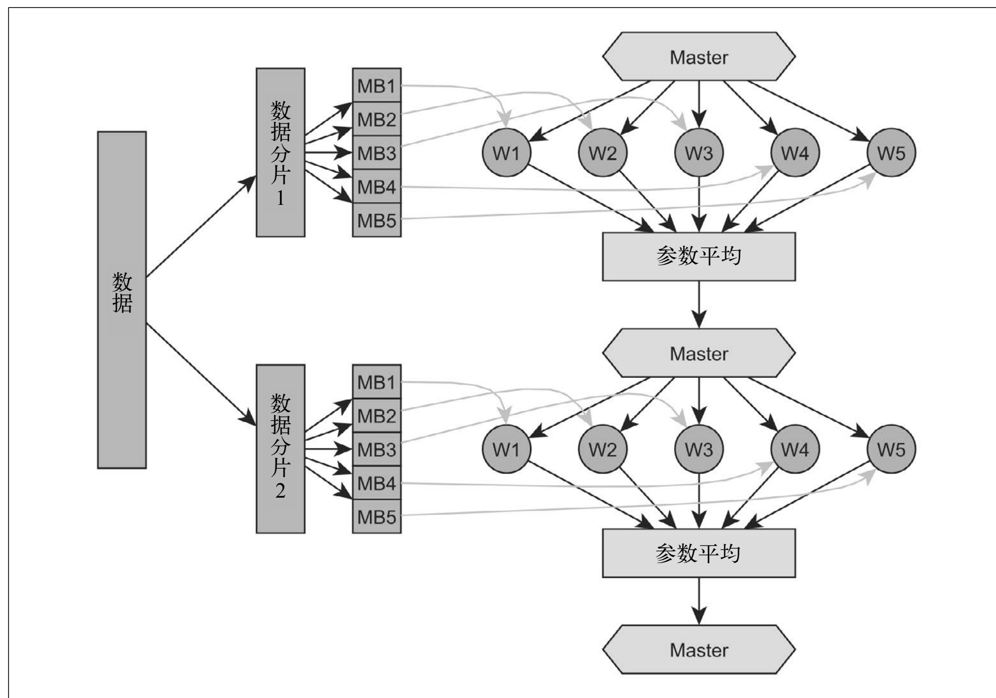


图 6-3：DL4J 的参数平均并行化策略

启发了 DL4J 并行化策略的主要研究论文之一是 Jeff Dean 和他在谷歌的团队介绍使用 SandBlaster 工具的论文。

Jeff Dean

Jeff Dean 是谷歌的一位杰出的研究工程师，他曾参与过谷歌一些核心系统的开发。Jeff 于 1999 年加入谷歌，并在谷歌知识集团担任高级研究员。正如刚才提到的，Jeff 领导了 DistBelief 和 SandBlaster（ICML 2012）的设计和实现。DistBelief 是谷歌训练深度神经网络的大规模分布式系统，已用于图像识别、语音识别、自然语言处理（NLP）和其他机器学习任务。以下是 Jeff 领导的其他有影响力的机器学习和分布式系统项目。

- 5 代爬行、索引和查询服务系统。
- 谷歌为内容创收设计的 AdSense 平台的早期开发。
- MapReduce 的设计与实现。
- BigTable 的设计与实现。
- Spanner 的设计与实现。

鲜为人知的事实：

Jeff Dean 直接用二进制写程序，然后他编写源代码作为文档提供给其他开发者。

DL4J 支持许多可用的运行时（例如亚马逊云服务、多线程、Hadoop 上的 Spark-YARN），但是它们都使用参数平均策略（就像现在一样）。正如前面提到的，这种技术在并行迭代研究文献中很常见，而且我们围绕移植到每个运行时的一组通用模式来设计。



参数服务器

参数服务器的设计正在被添加到项目中，但是目前参数平均依然是解决许多并行深度学习建模问题有价值的方法。

并行SGD执行

并行 SGD 执行模式的高级别执行流程如下：每个 worker 得到一个总数据集的分片，并在分到的记录上运行自己的小批量训练算法。在每个小批量的处理中，worker 将其参数向量发送回 master 节点，然后在处理流的**超级步骤**阶段，执行参数平均。



控制参数平均

在 DL4J 中，参数平均的频率是可配置的，但对“每个小批量”进行参数平均是低效的。在实践中，设置为“每 5 到 20 个小批量”执行一次表现很好。

然后，master 节点将这个新的全局参数向量发送回 worker，以便它们可以在数据分片内执行下一个小批量处理。这个过程一直持续到完成对数据集中所有记录的遍历或者完成对数据集的 *N* 次遍历（更多相关信息，请参阅介绍 Spark 的第 9 章）。

我们用一个并行训练的例子来说明这一点。假设有 10 个并行的 worker，每个 worker 有 100 条记录的分片（为简单起见）。然后每个 worker 将分片进一步细分成含 10 条记录的小批量。worker（并行地）将在每个小批量上执行学习过程，直到完成整个 100 条记录的训练，或者说完成它们分到的整个数据集的“分片”的训练。并行化带来的好处是：尽管

worker 学习得相对缓慢，但使得整体的训练速度大大加快。各种文献和开源实现中的实验结果显示，随着机器数量的增加，速度的改进近似于线性增长。

在实践中，并行训练和串行训练得出的训练效果可能不同。

在处理并行训练时，我们关注的两个主要的超参数是小批量大小和正则化。需要更细致地考虑这两个并行训练的超参数，因为它们的行为和在串行训练中不同。

- 小批量

  举一个并行训练的例子：当要处理的小批量过大，小批量中会出现数据集的离群值，这种情况会造成训练收敛周期变长，因为学习过程在较大的小批量中更难从离群值中获得信息。

- 正则化

  另一个例子是求解具有更多方差（小的正则化常数）的问题可以从并行化中得到更多模型。与串行方式相比，使用参数平均训练的主要区别在于减少了 I/O 开销以及利用数据局部性的能力（在将计算 worker 移动到数据块主机位置的实现中）。另一个副作用是参数平均表现得像是参数向量的正则化函数。

6.11.3 GPU

参数平均并不是加速迭代类算法的唯一方法。接下来介绍一个基于使用 GPU 的方法，它可以在其专用硬件上使用向量化的数学实现。如今市面上桌面 PC 显卡的最大内存带宽通常比现代 CPU 高出数倍。



现代 GPU 的 CUDA 内核超过 1000 个。

GPU 的一个有趣的特性是，它们可以与成千上万的线程同时工作，并且能够以很少的开销调度内核上的线性代数计算。这类细粒度的并行性使得 GPU 成为加速机器学习中大规模线性代数计算的一个有吸引力的候选。

我们希望以允许更大并行性的方式执行内存访问。GPU 提供了这样的一种方式，合并内存访问，并且硬件可以在有效访问速度比普通 CPU 和 RAM 场景快几倍的情况下并行执行内存访问，这使得 CPU 的 RAM 和 GPU 的 RAM 之间的数据传输成为主要瓶颈。在大多数情况下，这种传输时间甚至支配着 GPU 上的大型矩阵 × 矩阵计算。（为了对这个瓶颈有更多认识，来看一个例子：在 1000 × 1000 的矩阵乘法中，矩阵运算只占用 0.5% 的时间。）如果一次将更多的数据传送到 RAM 中，并且以更大的批量执行多个操作，那么可以从硬件和 GPU 中获得更高的效率。

可以在附录 C 中看到，当访问硬件时，批量执行内存操作。在将数据高效地传输到 GPU 硬件之后，就可以在这一次性地发送到 GPU 的训练批次内的不同数据块之上，利用数千个额外线程来操作。这些是 DL4J 中线性代数架构的主要驱动因素，我们通过它们使用批处理来加速大型神经网络中大量参数的计算。为了进一步加速这些计算，我们创建了

ND4J，使向量化计算能够在不同系统和不同 GPU 的线性代数运行时之间方便移植。



ND4J：一种可适应的执行引擎

我们从一开始就考虑到 DL4J 在 CPU、GPU 或并行框架上运行的需求。可替换的 ND4J 后端的灵活性相当大，提供了执行时的选择。

要了解更多将 GPU 设置为 ND4J 后端的信息，请参见附录 E。

6.12 控制迭代和小批量的大小

第 2 章介绍了小批量的概念。实践表明，将训练输入数据集分成小批量能够更高效地训练网络。一个小批量往往是从 10 个输入向量到整个输入数据集范围内的任何数据集。

这种方法还允许以向量化的方式执行某些线性代数运算（特别是矩阵－矩阵乘法）。如果有 GPU，也可以选择将向量化计算发送给 GPU。

下面是几个关于控制训练的关键术语。

- 轮

  一轮训练指整个输入数据集的完全传递。在训练收敛之前，对数据集进行多轮训练。

- 小批量大小

  小批量大小是一次性传给学习算法的记录（或向量）数量。与之相对的做法是一次传递一条输入记录去训练。

算法学习模型所耗费时间的曲线通常呈 U 形变化（批量大小与训练速度）。这意味着一开始随着批量的大小增加，训练时间将减少，之后当批量大小超过某个较大的值时，训练时间将开始增加。



随着小批量大小的增加，更多的计算意味着梯度可能更平滑，但计算成本更高。

理想情况下，每个小批量训练集应该包含每个类别的一个样本，以便在估计整个训练集的梯度时减少采样误差。

调整小批量大小

原则上（通过适当的调优），神经网络可以用任意大小的小批量去学习。但是在实践中，确定小批量大小时需要考虑以下因素：

- 内存需求
- 计算效率
- 优化效率

前面的章节已经介绍过内存需求了，所以这里不再讨论它们。

至于**计算效率**，DL4J（以及其他现代深度学习库）在数学运算，如矩阵乘法和向量运算（加法、对应元素的乘法等）的层面并行化学习。这意味着过小的小批量会导致硬件利用率不佳（特别是在 GPU 上），而过大的小批量可能导致效率低下——同样在小批量的所有样本上平均梯度，这意味着最终增加的更多梯度的计算成本超过了带来的好处。

关于**性能**（对于 GPU 来说这是最重要的），应该以 32 的倍数为批量大小，否则应该使用 16、8、4 或 2 的倍数（否则会因其他需求导致变化太大）。这样做的原因很简单：与其他大小相比，内存访问和硬件设计对于维数是 2 的幂的数组，操作优化得更好。



也应该考虑以 2 的幂的数字为层大小。例如应该使用 128 层而不是 125 层，或者使用 256 层而不是 250 层，以此类推。

关于**优化效率**，需要注意的是，不能完全独立于其他超参数，来选择小批量大小，例如学习率。较大的小批量意味着更平滑的梯度（即更准确 / 一致的梯度），辅之以适当的调优，就能更快地学习给定数量的参数更新。当然，它带来的不足是每个参数更新需要更长的时间来计算。使用更大的小批量可能有助于网络在一些困难的情况下学习，如有噪声或类别不平衡的数据集上。

那么应该怎样选择小批量的大小呢？在实践中，32 至 256 常用于使用 CPU 训练的情况，32 至 1024 常用于使用 GPU 训练的情况。通常，这个范围内的一些值对于较小的网络来说已经足够好了，不过你也许应该使用较大的值来测试（其中训练次数可能有限制）。

然而，内存需求最终会限制大型网络的最大批量的大小。



对于分布式训练，在共享硬件的每个执行器上使用较小的小批量并不罕见（比如在 Spark 上为每台机器使用多个执行器进行训练）。

最后不要忘记，当小批量大小增加时，每轮的参数更新数量减少了（一轮指的是训练数据的一次完全传递）。每轮的参数更新的数量只是训练集中样本总数除以小批量的大小。



小批量大小与轮的关系

为了保持相同数量的参数更新，如果小批量大小增加一倍，那么轮数也需要增加一倍。

6.13　如何使用正则化

某些类型的正则化有助于降低那些变得过大过快的参数向量的权重（例如 L1 和 L2）。在某些场景或机器学习教材中，权重衰减被当作正则化。其他类型的正则化，如 Dropout 和 DropConnect，除了减少大的权重之外，还使用其他方法来减轻过拟合。通常在训练中使用正则化方法，因为它不仅仅可以防止过拟合。

正则化能够使用不同的方法来防止参数值变得太大，从而高效地表示模型。找到用于正则化设置的正确组合通常通过手动调优完成，但也可以通过“随机搜索”或“网格搜索”来完成。稍后将简要介绍每种正则化方法及它们是如何影响整个训练过程的。

6.13.1 使用先验函数正则化

应用先验函数是正则化参数向量一种常用的机器学习技术。权重衰减通常通过正则函数如 L1 或 L2 先验函数来执行。在某些情况下，这些函数被组合，这对深度网络来说很常见。表 6-8 显示了如何根据正在建模的输入数据类型使用不同的先验函数。

表6-8：使用先验函数的场景的总结

先验函数名	使用场景
L1	稀疏模型
L2	密集模型

在文献中，L1 和 L2 正则化方法可同时用于每个函数的单独设置。对于使用早期停止的情况，不会使用 L2 正则化，因为早期停止在执行与 L2 相同的机制时更高效。L1 正则化已被证明是用作特征选择的有用形式。

在实践中，L2 正则化使用更频繁。以下是 L1 和 L2 的不同点。

- L2 更多地惩罚大的权重，但不会将小的权重变为 0。
- L1 对大的权重惩罚较小，但会导致许多权重变为 0（或非常接近 0），这意味着得到的权重向量可能是稀疏的。

可以将 L1 和 L2 结合在一个网络中。



在实践中，除了显式的特征选择之外，L2 正则化 L1 正则化表现更好。

稍后将介绍，所使用的数据也会影响建模的结果。

6.13.2 最大范数正则化

在这种形式的正则化中，为每个单独的隐藏单元设置传入权重向量的 L2 范数的上界。这与惩罚整个权重向量的平方长度形成对比，就像通常用 L2 范数所做的那样。

使用基于约束而非惩罚策略的方法，使得无论传入的权重更新有多大，都可以防止权重过大。

相比于其他使用较低学习率的方法，最大范数正则化从高的学习率开始，再加上学习率衰减策略（例如 AdaGrad），使得我们能够更完全地搜索权重空间。它已被证明在深度神经网络中能够很好地与 SGD 协同工作，即使没有 Dropout。

6.13.3 Dropout

Dropout 是一种强大的正则化方法，可以在许多类型的模型中使用。Dropout 是在模型训练过程中通过从网络中移除单元来正则化的，一种计算上成本不高的方法。Dropout 几乎能够在各种神经网络架构上工作，并能够很好地与 SGD 协同工作。Dropout 的工作方式是暂时将一个单元的激活值设置为 0.0。对于输入层神经元，Dropout 的概率被设置为 0.5 至 1.0 之间（例如保持或去除激活值的概率）。



有时根本不对输入使用 Dropout，特别是对于含有噪声或稀疏的数据集。

在隐藏层中，Dropout 的概率是 0.5。通过随机省略神经元，可以防止检测器之间的协同适应，这有助于从留存数据中得到泛化能力更好的模型。



输出层和 Dropout

在输出层上通常不使用 Dropout。

通常除了极端的设置以外，几乎所有的 Dropout 设置都会有帮助。通常将 Dropout 设置为 0.5，并且实践证明它在许多类型的网络和目标上工作得很好。在所有隐藏层中设置 Dropout 的效果比仅在一个隐藏层中设置的效果更好。即使不使用其他正则化技术，Dropout 也会使隐藏单元的激活变得稀疏，从而得出稀疏表示。

可以在 DL4J 中添加以下代码将 Dropout（以概率 0.5）应用到网络配置中。

```
.regularization(true).dropout(0.5).
```



Dropout 概率

在 DL4J 中，Dropout 概率（使用 `dropout(double)` 配置选项设置）是保持激活的概率，即 `dropout(0.7)` 意味着有 70% 的概率激活被保留，有 30% 的概率被设置为零。注意，在 DL4J 中，`dropout(0.0)` 表示“禁用此层的 Dropout”（因为保留 0% 概率的激活没有意义）。

根据 Srivastava 等早期的 Dropout 论文，Dropout 的影响并非特别取决于 Dropout 概率。在大多数情况下，它只要被设置为 0.5 就能很好地工作。但如果一个网络（相对于训练的数据量）非常大，应使用更小的保持激活的概率（这相当于更强的正则化）。

更多关于使用 Dropout 的说明。

- 在实践中 Dropout 通常与其他正则化方法相结合，如 L2 正则化。
- 作为一般的规则，应该避免在网络的第一层使用 Dropout（通过在该层设置 `dropout(0.0)` 来禁用它），这将避免删除输入数据集的重要部分。



DL4J、Dropout 与 RNN

对于 RNN，Dropout 通常只应用于进入层时的激活，而不是层内的循环激活。在循环激活上应用 Dropout 会使学习时间依赖变得过于困难。当应用于 RNN 时，DL4J 仅在输入激活时应用 Dropout。

Dropout的问题

研究发现当训练记录的数量上升到千万级别时，Dropout 并不那么高效。研究还表明，与不启用 Dropout 的相同网络架构相比，启用 Dropout 使训练时间增加了两到三倍。与标准 SGD 相比，Dropout 在学习梯度上的噪声也更多。

一种对抗梯度噪声的方法是使用更高的动量设置（从 0.9 提高到 0.95~0.99）。在某些情况下，这将导致权重变大，可使用最大范数正则化来控制它。



DropConnect

DropConnect 简单地将一些权重（暂时）设置为 0。DropConnect 与 Dropout 相关，但实际上它不是 Dropout 的变体，它们是不同的概念。

6.13.4 其他正则化事项

以下是其他一些需要了解的正则化事项。

- 随机池化

  随机池化是一个 CNN（专用的）正则化技术，通过它，我们使用一种随机池来构建 CNN 网络集。每个网络都参与每个特征在不同空间位置的映射。

- 对抗训练

  可以通过对训练数据集使用小却有意形成的格式错误的变体来创建输入，这将生成一个能够给出高可靠性错误答案的网络模型。实践证明这可以减少过拟合，并得到能够抵抗对抗样本的模型。

- 课程学习

  课程学习的做法是从小批量训练数据集中容易学习的样本开始学习，然后随着时间的推移，逐步训练更复杂的样本。以这种策略为网络的正则化器，能够同时使得训练更快和收敛更好，一举两得。



并行化与正则化

在学习期间，从参数向量的权重中去除噪声更好，部分原因是使用了学习率。从最终权重中去除一些噪声的一种方法是在几次更新中平均权重，也可以通过本章稍后将介绍的并行运行时模式中的参数平均来实现这一效果。

在某些并行化的情况下，不使用先验函数作为正则化器，因为参数平均本身达到了预期的效果。

6.14 处理类别不平衡

在机器学习中通常需要处理那些每个类别的数据数量显著不平衡的训练数据集。这种情况通常发生在我们最感兴趣的事件的发生频率远远低于事件没有发生频率的领域，例如预测是否有人点击网页上的广告。



PhysioNet 与重症病房死亡率预测

第 1 章提到的 PhysioNet 挑战数据集是具有挑战性的不平衡数据集的一个很好的例子。获取大量（非常）罕见的事件是一个挑战，如重症病房的死亡数据。

对于大多数学习方法，如果根据 99% 为负和 1% 为正的数据训练模型，那么模型将仅仅能学到总是预测占统治地位的类别（例如负）。一开始可能不会觉得有什么不对，因为平均正确分数会显得异常高，而且可能认为模型非常准确。最终的情况是：由于学习过程中的错误降到了最低，并且总是预测出值为正的结果，正确率可以达到 99%。

在不平衡数据的情况下构建模型的一个好的做法是不严格优化预测正确率。相反应该把重点放在评估模型上，如 F1 得分或曲线下面积（AUC）。我们需要模型预测一些值为正，但希望是在最合适的时候预测。

关于模型校准

对于类别不平衡的领域，网络经常会错误预测某些类别的不平衡。例如，当没有检测出患者患有某种严重的疾病、但患者实际患有这种病（假阴性）时，患者会因此死亡。在这个例子中，没有检测到致命疾病的后果比错误预测患者患有该疾病要大得多。从这个角度来看，更好地理解模型评估是现实中应用机器学习的关键。

为了解决模型评估问题，可使用一种称为**模型校准**的方法。模型校准通常基于一些改变值或概率的转换以更好地估计。表 6-9 总结了这些校准技术。

表6-9：校准问题的分类

类型	任务	问题	全局/局部	校准对象
CD	分类	期望类别分布不同于实类分布	全局或局部	预测
CP	分类	正确猜测的预期 / 估计概率与实际比例不同	局部	概率 / 可信度
RD	回归	期望输出不同于实际平均输出	全局或局部	预测
RP	回归	期望 / 估计误差置信区间或概率密度函数太窄或太宽	局部	概率 / 可信度

正如论文“Calibration of Machine Learning Models”（Bella 等人著）中所指出的，CD 和 RD 类型需要修改预测，以便校准结果。虽然本书没有详细讨论这些方法，但要注意这一课题的进一步研究。

在训练过程中有两种主要的方法来处理类别不平衡。

- 从较大类别中抽样或从较不频繁的类别中多采样。
- 基于加权损失函数对训练实例加权。

下面详细介绍这两种方法。

6.14.1 类别采样方法

处理类别不平衡的第一种方法是对较大的类别进行采样，这样就得到了一个平衡的训练数据集（例如 50% 为正，50% 为负）。这种方法的缺点是可能丢弃了好的数据，但好处是使得训练更高效。

还有一种方法是**事后缩放**。做法是正常训练网络，但在训练之后缩放输出。这种方法与其他方法的结果不太一致。

另一种方法称为**概率采样**。做法是首先随机（例如通过某个概率）选择一个类别，然后在该类别中随机选择一个样本。

最后一个方法是**超采样**，做法是复制低频类别，对它们超采样，直到达到其他类别样本的数量[4]。

6.14.2 加权损失函数

在训练期间处理类别不平衡的另一种方法是使用较小的权重来训练大类别中的样本。这有助于不平衡数据集的训练影响达到平衡，创建泛化能力更好的模型，以便检测更罕见的标签。

为了详细讲解这个概念，用一小段代码来说明。在下面的代码片段中，创建了一个 `INDArray`，它在加权损失函数中保存用于类别加权的类别权重。

```
INDArray weights = Nd4j.create(new double[]{0.1, 0.4, 1.0});
```

然后将这些权重传递到损失函数，如以下代码片段中突出显示的。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(1)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.learningRate(learningRate)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new DenseLayer.Builder().nIn(numInputs).nOut(numHiddenNodes)
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.RELU)
    .build())
.layer(1, new OutputLayer.Builder(LossFunction.NEGATIVELOGLIKELIHOOD)
    .weightInit(WeightInit.XAVIER)
```

注 4：在多轮训练中使用这些取样方法，对于深度学习来说每轮的取样结果之间并没有太大的实际差异，但对于 SVM 和随机森林而言，每轮的取样结果之间存在很大差异。

```
        .activation(Activation.SOFTMAX)
        .lossFunction(new LossNegativeLogLikelihood(weights))
        .nIn(numHiddenNodes).nOut(numOutputs).build())
    .pretrain(false).backprop(true).build();
```

使用加权损失函数背后的动机是改变优化目标，以便更严厉地惩罚对不常见类别的错误预测。理想状态下，这会使网络避免出现总是预测最常发生的（多个）类别的较差局部最优解的情况。



理解相对的权重值

1.0 的权重相当于没有加权（例如 {1.0, 1.0, 1.0}）。在这个配置中，所有值均为 1.0 的情况与根本没有提供权重数组的情况完全相同。对经常出现的类别应使用更小的权重。

6.15 处理过拟合

过拟合指机器学习工作流对训练数据集学习得过好，而对未接触过的数据集表现不佳的现象。这样的模型不能很好地泛化到更大的数据集。

如果模型在训练数据中学到偶然出现的模式，那么模型在测试用例之外的数据上将表现得很差。如果训练模型对输入数据学习得过好，那么它就不能很好地处理测试用例之外的数据，模型就不会像想象的那样有用。这两种情况都过拟合了训练数据，并且模型已经学到了随机相关性，这些随机相关性无益于模型对未接触过数据进行评分。



更好的泛化

机器学习中使用模型的目标是泛化训练数据集中包含的信息，以便该模型能够很好地处理来自相似源的更多数据。

所有的机器学习工作流在某种程度上都有过拟合的倾向，关键在于知道何时停止训练，使模型能够很好地泛化，同时尽可能使过拟合的量最小。当用深度学习网络对更复杂的数据集建模时，网络中总是使用更多的参数。需要权衡的是：必须以一定的速率增加参数以便对更复杂的数据集建模，同时速率又不能过快，以免引入了不必要的过拟合。

为了检测过拟合，可以在留存的测试集上评估模型的表现。重要的是，这个测试集不用于训练，否则网络在未接触过的数据上的执行情况就不得而知。至于修复过拟合的策略，可以使用以下一个或多个方法。

- 增加正则化（L1、L2、Dropout、DropConnect）。
- 早期停止。
- 更大的训练数据集。
- 更小的网络。

正则化通常是处理过拟合的第一步。L1 和 L2 正则化是有用的（之前探讨过）。Dropout 是一种常用且高效的将神经网络正则化的方法。



过拟合规则与参数数量

当模型中没有足够的参数时，它是不准确的。如果模型参数过多，模型在训练数据上的准确度看上去会很好，但模型会过拟合，留存数据上的准确度会偏离训练数据上的准确度。

6.16 通过调优UI来使用网络统计信息

DL4J 套件中的网络统计工具能够实时可视化和帮助我们理解神经网络实际在做什么。通过检查诸如激活、梯度和更新之类事情的当前状态（和轨迹），我们可以识别网络训练 / 配置问题，并执行策略来纠正这些问题。

设置 DL4J 的调优用户界面很简单。

```
UIServer uiServer = UIServer.getInstance();
StatsStorage statsStorage = new InMemoryStatsStorage();
uiServer.attach(statsStorage);
int listenerFrequency = 1;
myNetwork.setListeners(new StatsListener(statsStorage, listenerFrequency));
```

有关如何配置 `StatsListener` 以将统计信息保存到存储设备、更改 UI 端口或将结果发布到远程 UI 的详细信息，请参阅用户界面示例。注意，使用 `listenerFrequency` 设置收集信息（激活、更新等）的频率。可以增加这个数值以减少与收集统计信息相关的开销。每迭代 10 次收集一次统计信息是合适的，也可以将频率设置得更低（如果只是想在较长的时间段内跟踪调优良好的网络）。

DL4J 的训练 UI 有多个页面，其中第一个是概览页面。概览页面包括 4 个部分，如图 6-4 所示。

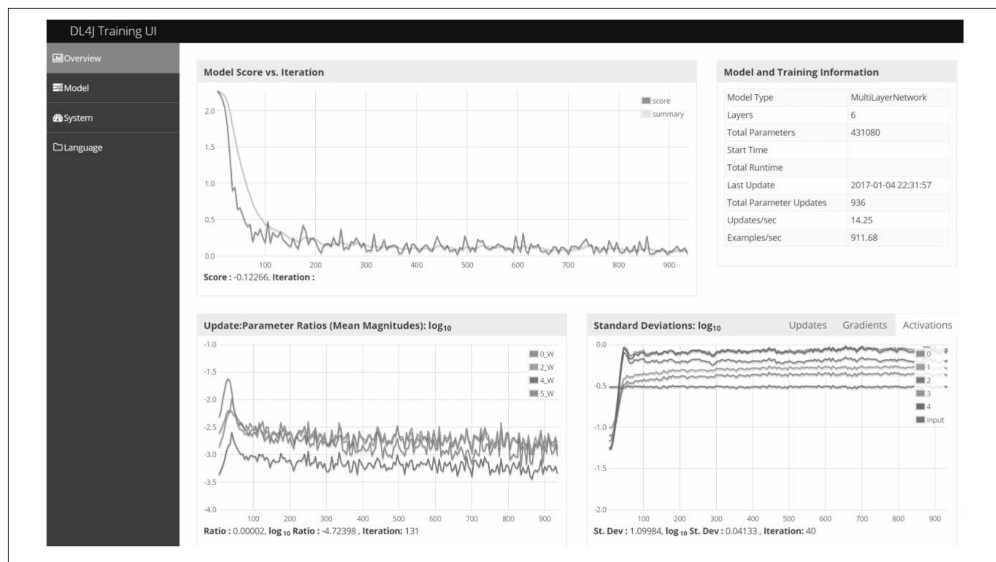


图 6-4：DL4J 网络统计工具

以下是概览页面提供的内容。

- 左上角：模型得分与迭代（得分的移动平均值）
- 右上方：网络信息概要
- 左下角：更新：参数比
- 右下角：激活、梯度、更新与时间

这里的“得分”指损失函数（MSE、负对数似然等）的值，以及添加到损失函数的任何正则化项（例如 L1、L2）。UI 的第二页是模型页，如图 6-5 所示，这样就能将模型结构可视化并分别获得每层的更多细节。

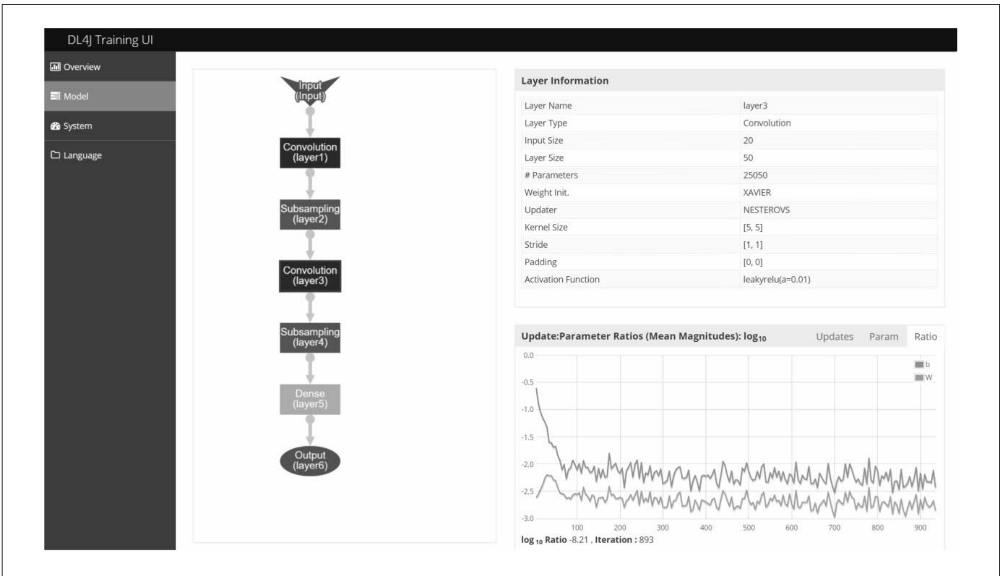


图 6-5：DL4J 训练 UI 模型页面

图 6-5 中的模型页被分成两部分，左边是一个网络结构的概要，显示了层（和顶点，在计算图的情况下），右边是一个图数量的汇总表（层配置），包括以下内容：

- 每个类型的参数随时间推移的更新比率（它还有参数平均大小和更新平均大小的选项卡）；
- 随着时间推移的层激活值（平均值和平均 ±2 标准差）；
- 层的每个参数类型的直方图；
- 层的（每个参数类型的）更新直方图；
- 随着时间推移的每个参数的学习率（注意：曲线将是平坦的，除非正在使用学习率调度器）。



关闭用户界面服务器

可以通过调用 `UIServer.getInstance().stop()` 关闭 UI，或者在完成之后简单地手动终止 JVM。UI 服务器将一直运行直至被关闭，即使训练已经完成。

最后，另一个有用的工具是计算测试集上的损失（或准确度），即不用于训练的数据。这对于识别过拟合更有用，而 UI 更多用于识别和修复优化困难的情况。通常先使用 UI 调整，然后添加测试集损失 / 准确度的计算。

6.16.1 检测不佳的权重初始化

简而言之，ReLU 和 Xavier 的初始化是基于网络架构（层大小）和关于激活函数的假设来设计的，它被用来完成两件事情。

- 确保网络激活的方差对所有层都是恒定的（即在网络后面的层中，激活不会增长得太大或太小）。
- 确保激活梯度（以及参数）的方差对所有层都是恒定的，即当梯度反向传播到前面的网络层时，它们不会增长得太大或太小。

但通常这两个要求无法同时满足。最值得注意的是，当相邻网络的层大小不同时（例如大小为 1024 的层的后面是大小为 10 的层，反之亦然），或者当激活函数与 Xavier/ReLU 初始化假设显著不同时（例如不是 ReLU/tanh/ 恒等函数，或者形状上不类似），可能需要使用不同的初始化。在深度网络中，这更成问题，因为它们可能只在有少数层的网络中通过次优的初始化而得以避免。

检测不佳权重的初始化有两种方法。首先可以在概览页面上使用 $(\log_{10})$ 激活值标准差图，如图 6-6 所示。

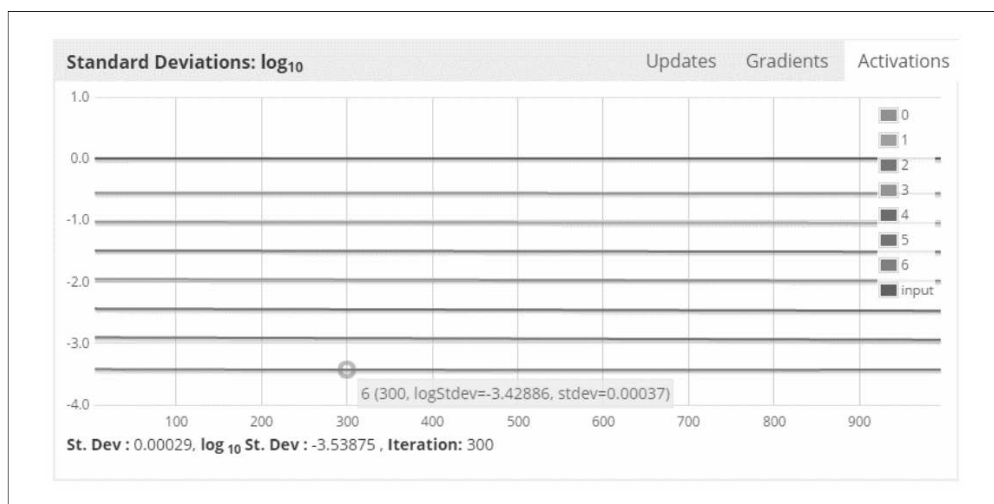


图 6-6：$(\log_{10})$ 激活值标准差图

这是采用 ReLU 权重初始化（带有 ReLU 激活函数）并将权重初始化方差减少至三分之一而产生的。通过这个（故意设置得较差的）权重初始化，激活逐层减少，直到在最后一层完全消失。在模型页面上，还可以通过查看层激活与迭代的图来检测这一点，如图 6-7 所示。

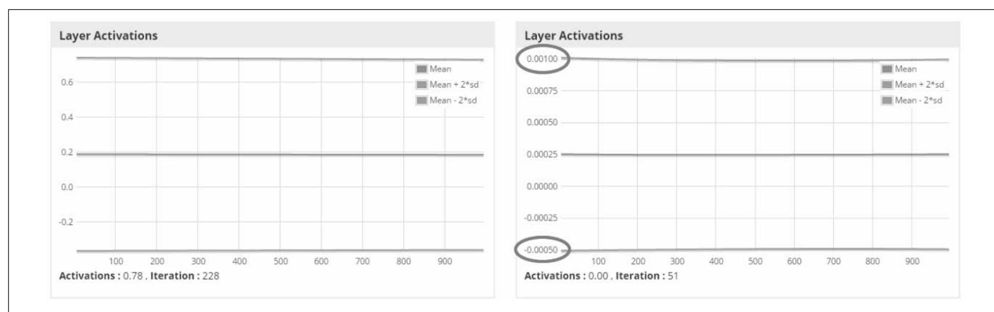


图 6-7：层激活图

如果权重初始化太大，激活也会变得太大。见图 6-7 所示的两个图表。



通常隐藏层激活值的对数标准差在 1.0 左右比较好。

注意，在解析这些图表时，不佳的数据规范化可能看起来与不佳的权重初始化有些相似。通常可以通过直接查看激活图来识别这一点，例如，在图 6-7 中可以看到输入标准差为 1，$\log_{10}(1)=0$。这是一个很好的输入量级，因此如果看到异常大或异常小的激活，应该检查数据规范化。

6.16.2 检测非混洗数据

通常使用小批量来完成神经网络的训练，小批量的定义是数据的子集（通常 32 到 1024 个样本）。为了在训练中取得更好的结果，数据需要被混洗。这意味着样本应该以随机顺序呈现，而不是（例如）同一小批量中前面的数据全部属于一个类别，后面的数据又全部属于另一类别。如果训练数据混洗得很好，训练应该如图 6-8 所示。

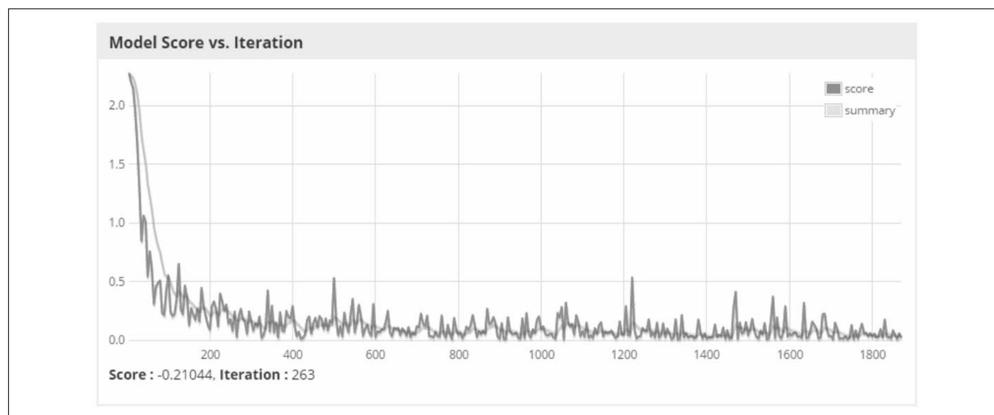


图 6-8：训练模型评分图的 UI 界面

为了说明不佳的混洗对训练造成的负面影响，考虑一下书中之前提到的 MNIST 数据集的情况。假设在 MNIST 上训练一个网络来预测数字，但不是按照随机顺序（如我们通常那样）呈现样本，而是首先按照标签数字的顺序对样本排序，这样首先只在 0 类别样本的小批量上训练，然后只用 1 类别的样本进行训练，以此类推。

图 6-9 表明，这种不佳的混洗方式会导致数据随着时间推移在模型得分中产生不同的模式。

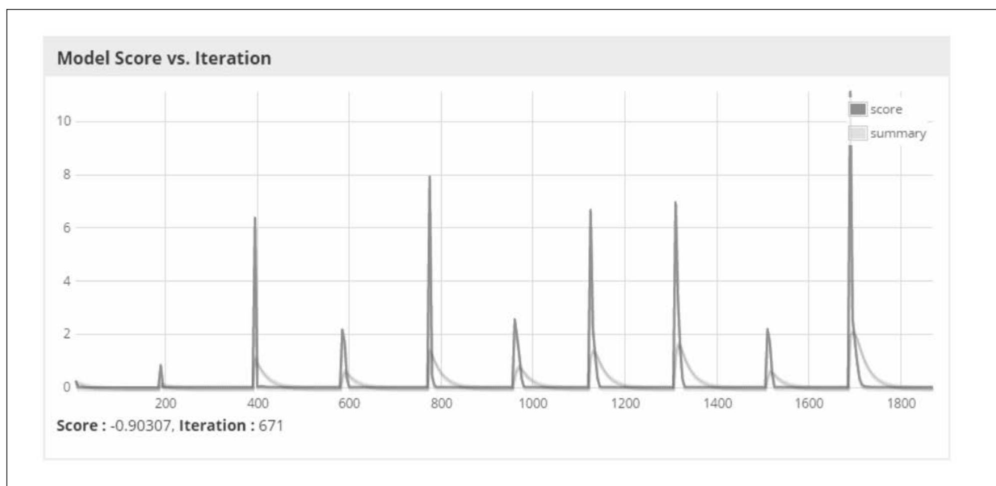


图 6-9：有峰值出现的训练模型分数图

在这里使用的模型通常会基于混洗数据适当地训练，并生成类似于图 6-8 的图。然而由于数据混洗不佳，训练分数中出现了周期性的大峰值，这与训练小批量中数字之间的切换相呼应。这里要说明的是，当出现这些周期性峰值时，我们最有可能对如何准备训练数据产生疑问。在数据准备的一系列工作中，这个问题通常容易解决，然后就可以开始了。



关于小批量中样本顺序的说明

同样值得一提的是，在一个小批量中混洗样本不会有什么差别。也就是说，如果一个小批量中有 32 个样本，那么这 32 个样本的顺序不重要，这是由于一个小批量中所有样本的梯度在应用于模型之前会取平均值。

6.16.3 检测正则化的问题

正如设置学习率一样，L1 或 L2 正则化参数值设置得太大或太小会导致训练结果质量较差。设置的值太大会导致权重变为 0，设置的值太小，又可能导致在某些情况下过拟合或学习不稳定。可以使用模型页上的 Param 选项卡来检测过大的 L1/L2 值（使权重为 0）。Param 选项卡展示了参数的平均大小，如图 6-10 所示。

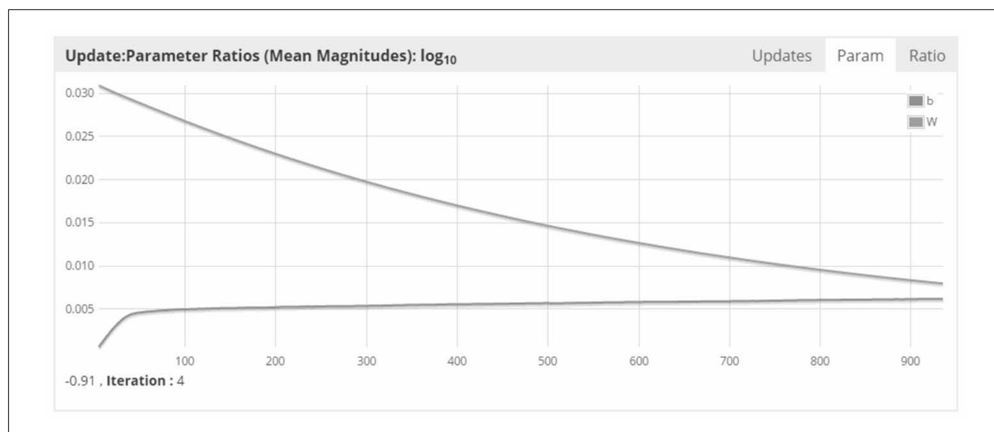


图 6-10：UI 上的权重值下降到 0

可以看到平均权重值开始较高，然后在迭代过程中变小，这表明正则化方法过于激进，L1/L2 正则化参数值应该降低。

可以用同样的工具检测的另一个问题是检测不充分的正则化。我们希望在层的参数直方图中看到权重呈现一个好的、正常的分布形状，如图 6-11 所示。

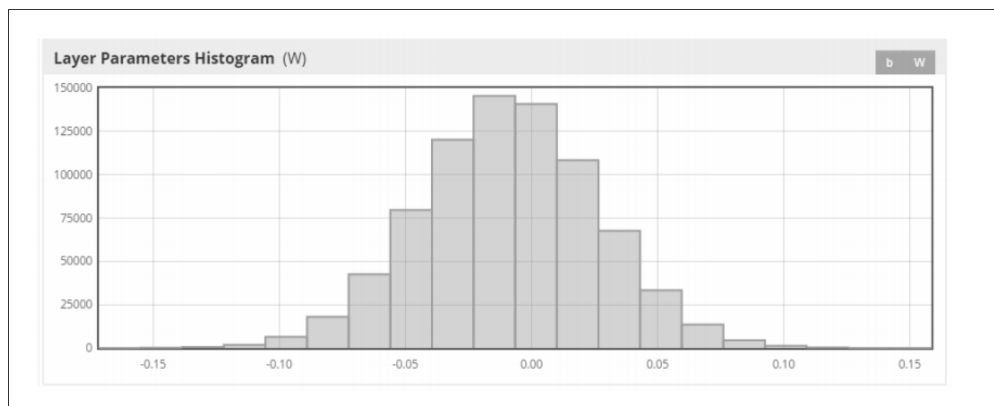


图 6-11：具有正常权重的层参数直方图面板

如果正则化不充分，少量参数常常会变得相当大。在这种情况下，层的参数直方图可能呈现为图 6-12 的样子。

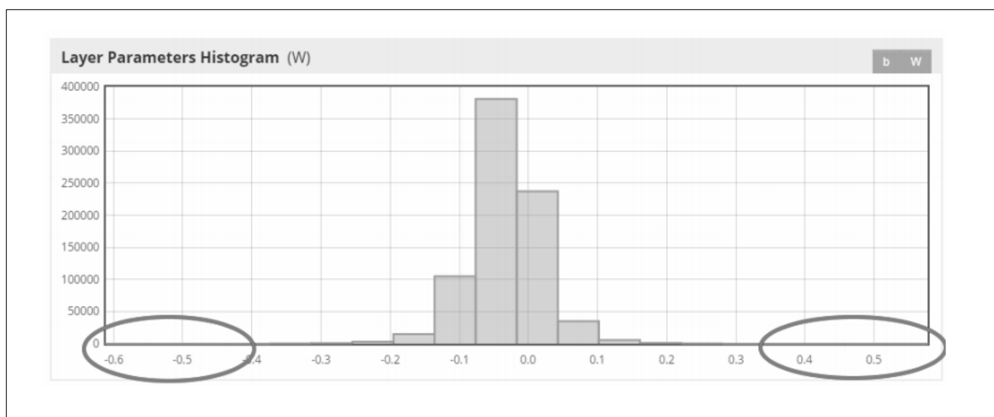


图 6-12：具有少量大的权重的层参数直方图

在图 6-12 中，可能很难看出发生了什么，因为在这个实例中只有少数参数在变大（注意柱状图中被圈住的很矮的部分）。在极端情况下，随着训练的进行，最大的参数可能向 $\pm\infty$ 发散。

第7章

调优特定的深度网络架构

> 你现在真正需要知道的是，宇宙比你想象的要复杂得多，即使你从一开始就认为它非常复杂。
>
> ——道格拉斯·亚当斯，《银河系漫游指南》

本章在第 6 章学习的一般深度网络调优概念的基础上，深入研究如何对特定的架构调优，例如：

- CNN
- RNN
- DBN

鉴于计算机视觉是深度学习领域中比较流行的应用之一，我们首先学习对 CNN 架构调优。

7.1 CNN

CNN 有一些通用的设计模式，以及特定的卷积架构设计模式。第 6 章介绍了通用的网络设计模式，本节将回顾与 CNN 架构相关的技术，重点介绍卷积层和池化层的布局模式。

从输入到网络层，卷积阶段使用多个过滤器学习不同的特征，如第 4 章所述。这个阶段的输出由 ReLU 激活函数转换。



检测器阶段

一些 CNN 文献将 CNN 架构中的检测器阶段划分为单独的层或阶段。检测器阶段仅仅是一个层的激活函数，在 DL4J 中，激活函数被视作层的一部分。本书将检测器阶段作为卷积阶段的一部分。

池化层通常插在连续的卷积层之间。在卷积层之后使用池化层，以逐步减小数据表示的空间大小（宽度和高度），这逐渐减少了网络中参数的数量和所需的计算。以这种方式减少参数也有助于避免过拟合。大多数与调优相关的其他主题，如输出层策略，第 6 章已经介绍过。

7.1.1　卷积架构常见的模式

在池化层之间常有一个卷积层，模式如下所示：

- 输入层
- 卷积层
- 池化层
- 卷积层
- 池化层
- 全连接层
- 另一个（可能的）全连接层

有时在池化层之前会有多个卷积层形成的序列。



在更大的网络中，在每个池化层之前堆叠两个卷积层是一个好的做法。这使得更大的网络在用池化层下采样之前，能够发现更复杂的特征。

为了进一步理解架构模式，回顾一下第 5 章介绍的 LeNet 示例。该网络最早在 1998 年由 Yann LeCub 公开，是最著名的卷积架构之一。它已被证明能够成功地对 MNIST 建模，正如 5.5 节介绍的那样。下面是一个特定层架构的列表。

- 输入层
- 卷积层：20 个过滤器 [5 × 5]
- 最大池化层：[2 × 2]
- 卷积层：50 个过滤器 [5 × 5]
- 最大池化层：[2 × 2]
- 全连接层

示例 7-1 展示了第 5 章 Java 代码示例中该架构的配置。为了简洁起见，这里只列出了代码的配置部分，便于比较刚刚描述的架构布局与特定的 LeNet 配置。

示例 7-1　Java 中的 LeNet DL4J 模型配置

```
/*
    构建神经网络
 */
log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
        .seed(seed)
        .iterations(iterations) //如上训练迭代
        .regularization(true).l2(0.0005)
        .learningRate(.01)
        .weightInit(WeightInit.XAVIER)
```

```
            .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
            .updater(Updater.NESTEROVS).momentum(0.9)
            .list()
            .layer(0, new ConvolutionLayer.Builder(5, 5)
                    //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的过滤器的数量
                    .nIn(nChannels)
                    .stride(1, 1)
                    .nOut(20)
                    .activation(Activation.IDENTITY)
                    .build())
            .layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                    .kernelSize(2,2)
                    .stride(2,2)
                    .build())
            .layer(2, new ConvolutionLayer.Builder(5, 5)
                    //注意nIn无须应用到后面的层
                    .stride(1, 1)
                    .nOut(50)
                    .activation(Activation.IDENTITY)
                    .build())
            .layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
                    .kernelSize(2,2)
                    .stride(2,2)
                    .build())
            .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                    .nOut(500).build())
            .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
                .NEGATIVELOGLIKELIHOOD)
                    .nOut(outputNum)
                    .activation(Activation.SOFTMAX)
                    .build())
            .setInputType(InputType.convolutionalFlat(28,28,1)) //查看下面的说明
            .backprop(true).pretrain(false).build();
```

在示例 7-1 中，每个卷积层后面都跟有一个池化层，之后网络进入一个全连接的 DenseLayer，接着是一个含 softmax 激活函数的输出层。这里的 softmax 激活函数用于检测训练 / 测试数据集的数据是 10 个 MNIST 数字中的哪一个。需要注意这里还有一种方法：

```
.setInputType(InputType.convolutionalFlat(28,28,1))
```

这个调用做了以下几件事情。

- 添加预处理器，它处理卷积 / 子采样层和密集层之间的转换之类的事情。
- 进行一些额外的配置验证。
- 必要时，根据前一层的大小设置每一层的 nIn（输入神经元的数量，或 CNN 网络中的输入深度）值。
  - 它不会覆盖通过 InputTypes 手动设置的值。
  - 可以把它用于其他类型的层（RNN、MLP 等），而不仅仅是 CNN。

通常在设计卷积架构时，应该使用多个较小的卷积层，而不是使用一个含有大感受野的单个卷积层。这使网络能够通过更具表现力的特征，从数据中提取更多非线性动态。

对于第一个卷积层，输入深度（nIn）必须与数据匹配，但通道数（nOut）是自由参数。“单元数”通常指 nOut。理想情况下，这个输入层的大小可以被 2 除多次，以便有足够的信息随时间下采样并构建特征。例如用 CNN 对 CIFAR-10（第 4 章介绍过）建模时，输入层的大小为 32。大多数情况下，使用 ReLU 激活函数来实现卷积层。

输入数据通道由 .nIn() 方法控制，如下所示：

```
.layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的过滤器的数量
        .nIn(nChannels)
```

可以根据输入类型配置它，但它通常在单色的情况下为 1，在 RGB 的情况下为 3。当然，如果输入不是图像数据并且具有多维表示，那么这个数字可能大得多。



关于第一个卷积层大小的一些注意事项

通常，输入层单元的数量取决于步长 / 过滤器大小。

一个有趣的例子是 AlexNet：$224 \times 224 \times 3$ 的输入，以及 11×11 的内核（在当时是非常大的）。

在实践中，可以将数据裁剪或缩放到一个适合要使用的网络配置的大小。但如果可能的话，最好是裁剪 / 缩小（不要放大）。

除了第一个卷积层，还有一种重复模式：更多的卷积层后面跟着池化层。在一些架构中，比如 VGGNet，多个（如两个）卷积层会连续出现在池化层之前。

通常，在 CNN 配置中，往往首先设置较大的过滤器，然后逐渐减小过滤器的大小。



卷积层序列上没有银弹

写作本书时，并不存在能够完美适用于所有图像建模问题的神奇的卷积架构。建议从已知的成功架构（如 LeNet、VGGNet、Inception 或 AlexNet）开始，利用它解决你的问题。之后，你可以调整层序列和超参数，使用已知的架构作为“已知良好的解决方案的起点”。

输出层将遵循第 6 章介绍过的模式。

7.1.2 配置卷积层

设置卷积层的空间排列，然后选择过滤器的数量。

在卷积层中设置超参数的方式决定了输出空间中神经元的数量及其排列方式。以下是关键的超参数：

- 过滤器大小
- 步长
- 填充

例如用 5×5 过滤器构建卷积层，那么代码如以下所示：

```
ConvolutionLayer convLayer = new ConvolutionLayer.Builder()
    .kernelSize(5,5).stride(1,1).padding(2,2)
    .name("first_layer")
    .nOut(out)
    .biasInit(bias)
    .build();
```

在这个示例中，过滤器大小（.kernelSize）是 5×5，步长 (1, 1) 和填充 (2, 2) 参数都采用了整数参数。使用 .name() 方法将这个卷积层命名为 first_layer，并通过 .nOut(out) 方法调用设置过滤器的数量。层本身的输出比单元输出数量更复杂。稍后将计算输出空间的大小。

1. 设置过滤器的步长

对于步长来说，我们围绕空间维度（宽度、高度）分配深度列。随着步长增大，感受野的重叠变少，并且输出空间变得更小。之前的例子中已经为卷积层设置了这个参数，如下所示：

```
.stride(1,1)
```

这意味着每次应用过滤器时，过滤器将向右“滑动”一个输入网格单元和向下“滑动”一个输入网格单元（更多相关信息，请参阅第 4 章）。如果步长为 2，那么过滤器将一次跳跃或滑动两个像素。



步长值超过 2

在实践中超过 2 的步长设置不常见，尤其在过滤器较小时。应该彻底避免设置大于过滤器的步长。

随着步长值的增大，输出空间会变小。实践证明，在卷积层中较小的步长（例如 1）能够更好地工作。

这使池化层得以进行下采样工作，并使卷积层将专注于仅转换空间中的深度。

2. 使用填充

在某些架构的某些组件中（如 Inception），会保留输入空间的空间大小（之后会探讨细节），因此使用零填充控制输出空间的大小。在输入空间经常会使用零填充，这样卷积层就不会改变输入的空间维度。零填充也保留了边界附近的输入信息。

3. 选择过滤器的数量

每个过滤器可以在输入的训练数据中寻找不同的东西。随着数据的变化和复杂性的增加，逻辑上需要更多过滤器来捕获相关特征，图 7-1 展示了从网络中第一个卷积层学习的过滤器。

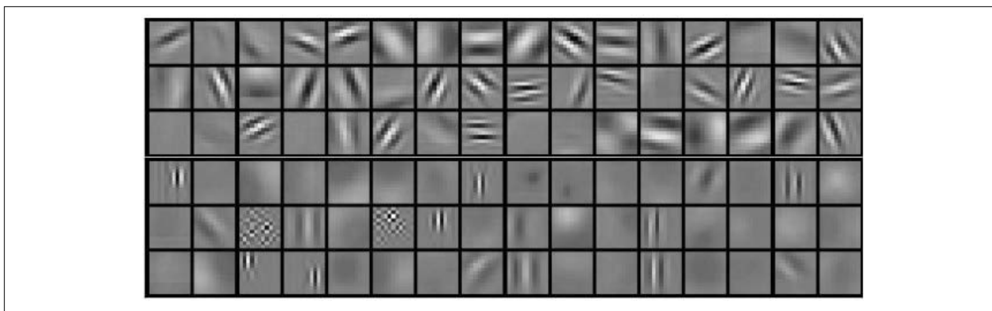


图 7-1：96 个学习过滤器的可视化，大小为 11 × 11 × 3

应该明智地选择卷积层过滤器的数量，因为计算单个卷积过滤器激活的成本比传统多层神经网络层的更高。在 DL4J 中，使用配置中的 `nOut(int)` 选项配置过滤器数量。

当网络沿 CNN 的层前进时，激活映射减小。靠近输入层的层的过滤器往往较少。越接近 CNN 的输出层，过滤器越多。

虽然没有特定的启发式方法来设置每个卷积层过滤器的数量，但需要了解，在一些较大的 CNN 架构中，过滤器的数量大约在 64 和 1024（高级网络）之间，并随着输入的增长而增长。表 7-1 给出了三种常用 CNN 架构的过滤器数量。

表7-1：过滤器数量示例

CNN	卷积层过滤器数量的增长
LeNet	20，50
AlexNet	96，256，384，384，256
VGGNet	64，128，256，512，512



计算成本与数据保存

所有层中的计算成本需要保持大致相同，因此每层的特征数量和像素位置数量的乘积需要大致相同。这意味着当从输入建立特征时，我们往往通过池化层下采样来丢弃输入数据。这使得直到网络的结束，输入相关的信息都能够保持，但是以不同的形式（特征）。

4. 配置过滤器大小

与较少的卷积层中的大过滤器相比[1]，多个堆叠卷积层中的小过滤器往往表现更好。

随着过滤器的增大，计算成本也随之增加，而且每个过滤器的输入区域也更大。空间过滤器较大（例如 5 × 5 或 7 × 7）的卷积，计算成本往往不成比例地高。例如对于网格中有 n 个过滤器的 5 × 5 卷积和有 m 个过滤器的 3 × 3 卷积来说，在过滤器数量相同的情况下，前者的计算成本是后者的 25/9 ≈ 2.78 倍。

注 1：Ciresan 在 2011 年引入了小过滤器的概念，但是网络并没有当时其他网络那么深。Goodfellow 在 2014 年应用了类似的方法来识别路标。GoogleLeNet (Szegedy) 也在 2014 年使用这种方法赢得了 ImageNet。

建议选择一个明显小于输入空间维度，但是又足够大、能够捕获相关特征的特征大小。对于卷积层，使用小的过滤器（3×3 或 5×5）和值为 1 的步长[2]。与之相比，GoogLeNet[3] 是一个相对复杂的 CNN，过滤器的大小在 1×1、3×3 和 5×5 之间。

另一个流行的网络 VGGNET 有一些有趣的特性，已被证明是成功的。卷积层过滤器的大小通常为 3×3，这是保留捕获像素（例如左、右、上、下和中心）概念的最小尺寸。



过滤器大小与连续层的比较

用三个具有 3×3 过滤器的卷积层加上池化层，类似于用一个具有 7×7 过滤器的卷积层。

添加更多的卷积层和过滤器使得在学习过程中能够利用更多的正则化。更多的层应用更多的非线性转换，并减少了被应用参数的数量。

另一种获得层序列和过滤器形状的方法是基于 VGGNet 或 GoogLeNet 这样的成功架构，从这些网络中观察到的模式[4]中汲取灵感。



v3 Inception 论文中关于过滤器大小的警告

我们注意到文中提到：

> 上述结果显示，过滤器大于 3×3 的卷积通常是没有用的，因为它们总是可以简化为 3×3 卷积层的序列。

5. 卷积模式与输出空间大小的计算

计算输出空间大小的通用公式如下，它是输入空间大小的函数。

输出空间的大小 = (W − F + 2P) / S + 1

表 7-2 定义了其中的变量。

表7-2：输出空间大小的变量

变量	描述
W	输入空间大小
F	卷积层神经元感受野的大小
S	步长设置
P	零填充设置

这个方程的结果应该是整数，如果不是，说明配置不正确。如果结果不是整数，可以使用填充、不同的过滤器大小，或者使用截断模式（稍后介绍）来修复它。需要注意：卷积层保持其输入空间的大小，而池化层通过下采样减小输入空间的大小。

注 2：例如 MNIST 输入图像的空间维度为 [28×28]，第一个卷积层上过滤器大小通常为 [5×5]。

注 3：也称“Inception”，是 2014 年 ImageNet 大规模视觉识别挑战赛 (ILSVRC 2014) 中分类和检测类别的冠军。

注 4：AlexNet 等网络中的过滤器 / 内核大小在现代标准下有点大。同样它可能需要被测试，并且请牢记：可以在处理图像（例如裁剪或缩放到某些尺寸）使之有助于网络更好地工作之后，设置图像大小。



跟踪输入空间

如果使用的步长大于 1，或者不对卷积层中的输入进行零填充，那么需要通过卷积网络跟踪输入空间。网络中所有的步长和过滤器需要保持平衡。

DL4J 确实有一些输入验证可以捕获许多无效的配置，然而要想知道如何修复问题（或者知道每一层中什么是有效的），仍然需要遍历整个空间。还需注意的是，有些配置是任何数量的填充都无法修复的。

在 DL4J 中，对于给定的输入大小和网络配置（特别是步长 / 填充 / 核大小），`ConvolutionMode` 定义了卷积层和子采样层应该如何执行卷积操作。

目前，DL4J 中提供了三种模式：

- 严格模式
- 截断模式
- 同一模式

在严格模式下，卷积层和子采样层在每个维度上的输出大小的计算方法如下：

$$输出大小 = (输入大小 - 核大小 + 2 \times 填充) / 步长 + 1$$

如果输出大小不是整数，那么在网络初始化或前向传递时会引发异常。

在截断模式下，卷积层和子采样层在每个维度上的输出大小都以与严格模式相同的方式计算。如果输出大小是整数，那么严格模式和截断模式相同。但如果输出大小不是整数，那么在截断模式下，输出大小将被向下舍入到整数值。



向下舍入的后果

向下舍入的主要后果是边界 / 边缘效应。实际上，沿给定维度（高度或宽度）的一些输入将不被用作输入，因此一些输入激活会被丢弃 / 忽略。在网络中（其中裁剪的激活可能在原始输入中占很大比例）这可能存在较大的问题，可能需要设置大的核和步长。

同一模式与严格 / 截断模式的操作方式不同，主要区别有以下三点。

- 不使用卷积层 / 子采样层配置中的手动填充值，而是基于输入大小、核大小和步长自动计算填充值。
- 与严格 / 截断模式相比，输出大小的计算方式不同（稍后将详细探讨）。最明显的是，当步长 =1 时，输出大小与输入大小相同。
- 对于顶部 / 底部和左部 / 右部，计算出的填充值可能不同（比如右部和底部可能比顶部 / 左部多填充一个像素 / 行 / 列）。

7.1.3 配置池化层

池化函数用相邻连接的神经元输出的汇总统计值代替特定点的层输出，这有助于模型表示在输入数据有微小变化时保持不变。池化层本身没有参数要处理，因为它们使用固定函数计算输入。零填充通常不用于池化层。



池化层的微小变化不变性

在训练图像数据时，模型对微小变化不变或对局部平移不变是有用的。

使用池化的最大原因是卷积层学习的函数必须对微小变化是不变的，使得通过网络的统计效果能够得到更强的卷积网络。

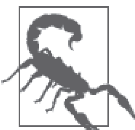
通常将池化层中进行下采样操作的最大池化的形状（例如空间维度：宽度、高度）设置为 [2 × 2]，如下面的代码片段所示：

```
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

需要注意：如果过快地增加池化空间的维度，就会面临丢失过多信息的风险。

在实践中，最大池化往往以下面两个池化层变体的形式出现。

- 感受野大小为 3，步长为 2（重叠池）。
- 感受野大小为 2，步长为 2（更常见的变体）。



如果池的感受野增加到超过了这些设置，往往会丢失太多的信息。

最近最大池化比其他变体更受欢迎，如平均池化或 L2 范数池化。与其他变体相比，最大池化在实践中表现得更好。

7.1.4 迁移学习

由于任务所需的处理能力 / 时间，CNN 模型很少从随机初始化参数开始训练。第 4 章介绍了深度学习流行的一大驱动因素是许多可用且高质量的大规模数据集，如 ImageNet（120 万已标记的训练图像）。虽然像 ImageNet 这样的基准训练数据集已经变得更好，但是我们很少基于一个大的标记训练数据集来使用 CNN 执行大规模的图像训练。改善这个问题的一个方法是从先前训练的结果良好的 CNN 模型开始，然后进一步在特定的图像数据集上训练，这被称为“迁移学习”。

1. 从零开始训练的替代方案

CNN 在网络的前几层学习了常见的视觉特征，然后在后几层逐步建立数据集的特征。这些早期的层特征类似于 Gabor 过滤器和斑点，并且大体上是视觉处理的“构建块”。以下是需要了解的迁移学习的两个用例。

- 对现有模型调优。
- 以现有卷积模型为特征提取器。

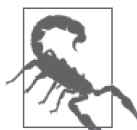
虽然这两种方法通常都使用 ImageNet 上的预训练模型，但它们的主要区别是最终的调优方式不同。下面是每种调优方式的一些细节。

- 对现有卷积模型的调优

  在这个迁移学习的变体中，CNN 模型在像 ImageNet 这样的大型数据集上训练，然后用特定于数据集调优的东西替换最后一个“分类器层”。有些变体将继续在所有层上执行反向传播，而其他变体则只更新后面的层。这是因为早期层的许多特征对于所有类型的视觉处理都通用，基本上不需要更新它们。后面的层专注于以任务特定的方式组合这些较低级别的特征，并且与针对特定领域的数据集的训练更加相关。

- 以现有卷积模型为特征提取器

  在这个迁移学习的变体中，采用最后一个全连接的层，即输出层，并且考虑把 CNN 的其余部分作为较小数据集的“特征提取器”。这也是相关的，因为 ImageNet 等预训练数据集在输出层中将有 1000 个不同类别的输出，并且可能与特定领域的任务不相关。



从零开始训练 ImageNet

在 ImageNet 上使用多个 GPU 训练 CNN 需要耗费两到三周的时间。对于实践者来说，发布他们的模型、供他人用作继续训练自己模型预训练的起点，是常见的做法。Caffe 项目有一个模型“动物园”（https://github.com/BVLC/caffe/wiki/Model-Zoo），可以从中下载许多预训练过的模型。

2. 何时考虑尝试迁移学习

可以在以下场景（或场景的组合）中尝试使用迁移学习。

- 训练数据集很小。
- 训练数据集与基本数据集共享视觉特征。

在其他场景中，你也可能成功应用迁移学习。



关于迁移学习中的学习率

在迁移学习中调优时，建议对较小的调优数据集使用较低的学习率，因为预先训练的权重可能已经很好了。

7.2 RNN

RNN 与其他架构（如 CNN）有一些相似之处，但是它们都有其自身的难点和超参数集合。与大多数神经网络一样，没有一种确定的方法能立即找到最佳的设置。超参数经常会拖延学习过程，甚至会造成网络比以前更糟。在此提醒大家，所有的网络都需要不断试错。



DL4J 与 LSTM 网络

DL4J 目前支持最流行的 RNN 变体：LSTM 模型。至于 RNN 的其他变体，DL4J 正在积极开发之中。

DL4J 还支持双向 LSTM。

RNN 的变体还有门控循环单元（GRU）和“普通”RNN。

7.2.1 网络输入数据和输入层

传统上，标准前馈神经网络的输入数据是单个向量（一维）或矩阵（二维，包含用于训练的小批量向量）。RNN 具有代表输入时间序列数据时间维度的输入数据的第三维。在 DL4J 中，使用以下参数表示输入（参见图 7-2）：

- 样本数
- 输入大小（列数）
- 时间序列长度

构造用于 RNN 的空间输入数据要比传统的网络耗费更多的精力，如多层感知器。正如先前书中讨论过的（如图 7-2 所示），输入有三个维度：

- 小批量大小；
- 每个时间步向量中的列数；
- 时间序列长度。

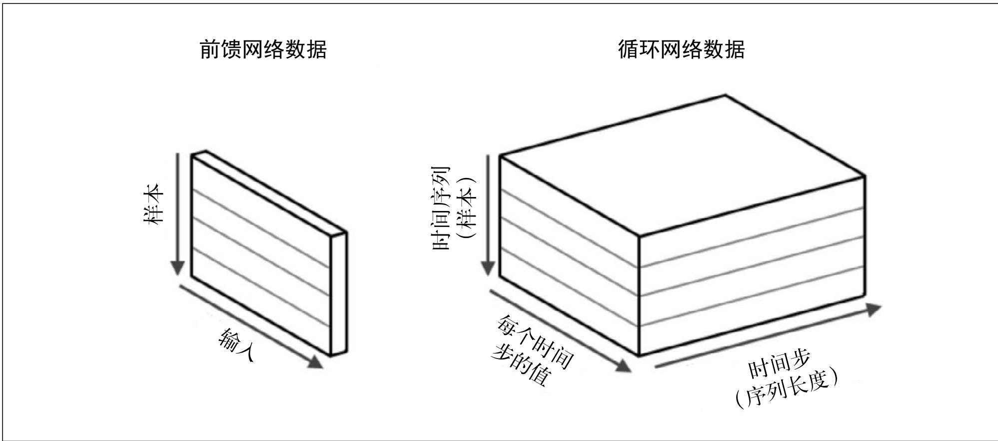


图 7-2：RNN 的三维输入表示

为了更容易理解是如何做到这一点的，请参阅附录 E 和附录 F。第 8 章将研究时间序列向量化的一般模式。



标准化

通常，对所有神经网络的输入数据（例如零均值、单位方差）进行标准化是有用的，这有助于将输入转换成更适合标准激活函数使用的范围。

标准化有助于输入和目标之间的关系保持尽可能地简单和本地化。记住，这仅仅用于实值输入，不要对独热（分类）输入这样做。

基于 RNN 训练和调优方式的不同，存在着一些基本的变体。

7.2.2 输出层与RnnOutputLayer

RNN 的输出有三个维度：

- 小批量大小（样本数）
- 输出的大小（列数）
- 时间序列长度

假设 DL4J 中的矩阵表示是基于 `INDArray` 类的，这些值与索引模式（i，j，k）相匹配，如表 7-3 所示。

表7-3：理解RNN输入矩阵中的变量

变量	说明
i	小批量中样本的索引
j	每个时间步中列的索引
k	时间步的索引

在 DL4J 中，许多 RNN 中以 `RnnOutputLayer` 为最终层，用于下列任务：

- 回归
- 分类

`RnnOutputLayer` 执行以下任务：

- 分数计算；
- 基于给定损失函数的误差计算（预测值与实际值）。

`RnnOutputLayer` 在功能上类似于 DL4J 中用于普通前馈网络的标准 `OutputLayer`，但它可以处理三维输出。`RnnOutputLayer` 的配置遵循与其他层相同的设计。对于分类，`MultiLayerNetwork` 中的最后一层设置为 `RnnOutputLayer`，如示例 7-2 所示。

示例 7-2　`RnnOutputLayer` 的配置

```
.layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
    .activation(Activation.SOFTMAX)
    .weightInit(WeightInit.XAVIER)
    .nIn(prevLayerSize)
    .nOut(nOut)
    .build())
```

为了更好地了解这一层的作用，请参阅第 5 章中 RNN 的示例，那个示例对传感器读数进行了分类。

7.2.3 训练网络

RNN 训练的计算成本高。通常情况下，SGD 是一种很好的适用于 RNN 的基线优化算法。研究表明，Hessian-free 优化也有助于训练 RNN。

1. 初始化权重

初始化权重在 RNN 训练中很重要。实践证明，对权重的良好初始化能够创建在足够长的距离上传递信息的隐藏单元，以便对长期依赖任务建模。

对于 DL4J 中的大多数 RNN 层，推荐使用 Xavier 初始化[5]，如下面的代码片段所示：

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
```

2. 时间反向传播

当循环网络处理含许多时间步的长序列时，基于时间的截断反向传播（BPTT）表现得很好。BPTT 是标准反向传播算法在 RNN 上的扩展（如第 4 章所介绍的）。截断 BPTT 降低了 RNN 中每个参数更新的计算复杂度。从下面的代码示例中可以看出，在 DL4J 中使用截断 BPTT 和 RNN 是很容易的。

```
.backpropType(BackpropType.TruncatedBPTT)
    .tBPTTForwardLength(100)
    .tBPTTBackwardLength(100)
```

这个添加到网络中的设置会使训练使用截断 BPTT。从代码中可以看出，前向和反向传递的长度分别通过各自的参数来设置。以下是在 DL4J 中使用它的其他注意事项。

- DL4J 将默认使用完整的 BPTT，除非指定截断 BPTT。
- 通常将长度选项设置在 50 到 200 个时间步之间（当然要看具体的应用程序）。
  - 前向和反向的时间步长度往往相同。
  - 反向时间步可以更短，但不能更长。

截断 BPTT 的时间步的长度不能超过输入时间序列。

3. 正则化

Dropout 是一种对 RNN 使用正则化的常用方法。在 RNN 中，Dropout 仅应用于 LSTM（非循环连接）中的连接子集，这种 Dropout 变体被证明是有用的。在 DL4J 中为 RNN 实现的 Dropout 仅对输入连接 / 激活使用，而不对循环连接使用。



标准 Dropout 与递归神经网络

标准 Dropout 已被证明不能很好地与 RNN 一起工作，因为循环放大了噪声，这会损害学习过程。

RNN 比多层感知器网络对学习率和动量的设置更敏感。

注 5：RNN 的正交权重初始化也显示出很好的前景。

7.2.4 调试LSTM的常见问题

虽然神经网络的调优需要大量试错，但是经过一段时间后，一些模式会有助于更快地对网络调优。有太多人试图通过盲目改变设置来调优网络，这不是最佳的做法。有时梯度会变得非常大（这被称为“梯度爆炸”，在更新的平均值图像上也能看到这一变化）。梯度爆炸的一个副作用是在多个小批量上损失函数的值会增加。这里真正的危险在于，对于大的梯度值，最终需要对参数做大的更改，这有可能破坏模型学到的特征。我们设置梯度裁剪或梯度再规范化来避免这一问题。梯度爆炸问题是普通 RNN 的一个常见的缺陷。

梯度爆炸和梯度消失

众所周知，RNN 除了“梯度爆炸”之外，还存在“梯度消失”问题。当梯度变得太小并且难以在输入数据集的结构中对长期依赖（10 个时间步或者更多）建模时，就会出现梯度消失问题。

解决 RNN 中梯度消失问题[7]最有效的方法是使用 DL4J 支持的 RNN 的 LSTM 变体。处理这一问题的其他方法包括：

- 在未折叠流的图（时延神经网络）中组合短路径和长路径；
- leaky 单元和不同时间单元的层次结构；
- GRU；
- 更好的优化方法；
- 梯度裁剪（用于梯度爆炸）；
- 促进信息流的正则化；
- 使用 L1/L2 惩罚。

7.2.5 填充与掩码

填充指将零（填充）添加到任何比小批量中最长的时间序列短的时间序列的末尾。这使得训练数据成为矩形数组（矩阵），当在同一个小批量中训练不同长度的时间序列时，这样的数组是训练所需的。掩码涉及两个附加数组，用于指示每个输入或输出最初是存在于输入中还是填充值中。

DL4J 中的填充和掩码支持以下循环模型训练的变体：

- 一对多
- 多对一
- 可变长度时间序列（在同一个小批量中）

图 7-3 给出了其中的每一个示例。

注 6：LSTM 有助于解决梯度消失问题，但不能从根本上解决梯度爆炸问题。

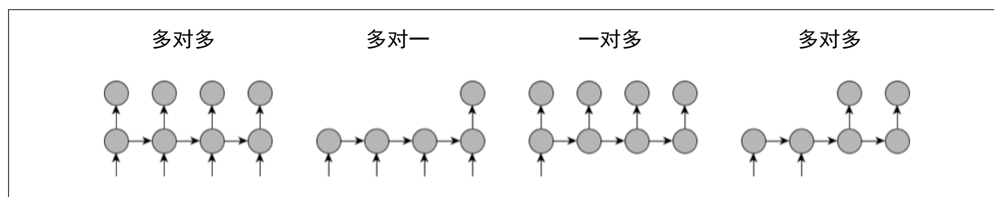


图 7-3：RNN 训练的不同变体

填充和掩码对含有不是在每个时间步中都出现的输入或输出的 RNN 是有用的。



填充和掩码的实用性

在没有掩码和填充的情况下，我们只能处理多对多的训练场景。这种场景中所有输入记录的长度都完全相同，并且在所有的时间步上都有输入和输出。

填充和掩码在空间输入中的应用

在 DL4J 中训练 RNN 涉及以下与小批量形状有关的参数。

- 小批量大小
- 输入尺寸
- 时间序列长度

填充数组包含以下用于输入和输出的参数。

- 小批量大小
- 时间序列长度

二维数组中每个值为 0 或 1，表示在填充之前的输入数据中，序列中给定时间步长最初缺失或存在。用于输入和输出数据的掩码数组是分开存储的。

如果使用所有值为 1 的掩码数组，那么它与根本没有掩码数组的效果完全相同。在多对一的情况下，只对输出有单一的掩码数组。在 DL4J 中，这些掩码数组是在数据导入阶段（如 `SequenceRecordReaderDatasetIterator`）创建的，并在 `DataSet` 对象中表示。DL4J 中的 `MultiLayerNetwork` 对象知道在训练时如何处理可能存在的掩码数组。

7.2.6　掩码评估与评分

在对 RNN 进行评分和评估时，也就是在评估模型的准确度时，使用掩码数组。在多对一的情况下，每个样本只有一个输出，在评估时需要考虑到这一点。

1. 利用评估类进行分类

在评估时使用输出掩码数组，将其传递给 `Evaluation` 对象，如示例 7-3 所示。

示例 7-3　评估设置

```
Evaluation.evalTimeSeries(INDArray labels, INDArray predicted, INDArray outputMask)
```

表 7-4 是对示例 7-3 中每个变量的说明。

表7-4：变量说明

变量	说明	维度
labels	训练数据的实际输出	三维
predicted	从网络生成标签	三维
outputMask	用于输出的掩码数组	二维

需要注意，在评估时不需要输掩码数组，因为只有网络的输出与评估有关。

2. 使用`MultiLayerNetwork`对新数据进行评分

也可以使用掩码数组计算模型的得分。得分计算不同于以前的做法，现在计算的是损失函数的值，而不是度量值，如准确度或 F1。不过在这两种情况下，应该对那些实际存在的时间步而非那些只是填充的时间步进行评分 / 评估。可以使用 `MultiLayerNetwork` 类来对时间序列数据进行评分，如下面的代码示例所示：

```
MultiLayerNetwork.score(DataSet)
```

正如前面 DL4J 中的掩码示例一样，如果 `DataSet` 包含输出掩码数组，它将被自动应用于计算网络的得分（损失函数）。

7.2.7　循环网络架构的变体

有些情况下，用户可能希望结合不同类型的架构层来对不同结构的数据建模，例如视频数据。在某些情况下，我们会结合 LSTM 和 CNN 层来对图像序列（例如视频帧）建模。DL4J 提供特殊的预处理器类来帮助建立这些架构的变体，如 `CnnToRnnPreProcessor` 和 `FeedForwardToRnnPreprocessor`。



你可以手动添加 RNN 的预处理器，但在许多情况下，它们将由网络自动添加。

7.3　受限玻尔兹曼机

受限玻尔兹曼机（RBM）是一种神经网络，通过将输入数据映射到隐藏状态，然后尝试从隐藏状态重建输入，以无监督的方式从数据集学习特征。在这一点上，它类似于无监督训练的其他模型，例如自动编码器（降噪、压缩和变分的变体），尽管 RBM（对比散度）的训练过程与自动编码器相比有很大不同。

通常 RBM 重点应用于以下领域。

- DBN 的无监督特征学习。
- 数据重建。
- 推荐引擎。



MNIST 实例

以学习过的 MNIST 数据集为例，网络的输入层有 784 个输入神经元，隐藏层有较少数量的神经元。在这个例子中，选择一个有 500 个神经元的隐藏层（大约是可见输入层的三分之二）。对于第二个隐藏层，设置神经元数量为 250 个。

下面考虑一下如何对 MNIST 之外的问题设置隐藏层神经元数量。

7.3.1　隐藏层神经元与可用信息建模

这里不使用判别式机器学习中的做法。在判别式学习中，训练实例约束模型参数，要求其数量等于指定标签所需的位数。



信息和样本标签

标签最多只保留少量信息，所以这不会带来巨大的工作量。在某些情况下我们使用比训练用例更多的参数，在所构建的模型中，这会产生严重的过拟合（有可能，取决于正则化）。

输入向量中的位数可能比用于表示标签的位数大几个数量级。从中可以看出，数据输入向量比标签有更多的信息要学习。

使用隐含在输入数据中的原始信息，我们可以利用这种结构的优势来更高效地提取特征或允许之后在管道中（当被用作 DBN 的一部分时）进行更高效的判别分类。我们想看看有多少信息可用。训练记录的信息量将影响隐藏单元数量的选择。



快速了解样本

训练图像数量：10 000。

每个图像的像素：1000。

投射的全局连接隐藏单元：1000。

可以看出，这个例子中隐藏（全局连接的）单元的数量与输入向量或图像的信息量直接相关。在没有全局连接或使用权重共享方案的 RBM 网络中，可以使用更多的单元。下面展示一个为 RBM 选择特定数量的隐藏单元的启发式做法。



隐藏单元数的选择

为 RBM 选择隐藏单元的数量时，要重点关心的是避免过拟合。在这个限制下，启发式的做法可以将可见单位的数量乘以 0.75 来得到隐藏单元的数量。如果稀疏目标很小，就可以避免使用更多的隐藏单元。对于输入数据集包含大量非常相似的输入记录的情况，也可以使用更少的参数。

7.3.2 使用不同的单元

开始接触 RBM 和 DBN 时，通常从 RBM 中的二元（或逻辑）单元开始。只靠这些单元的用例训练设置也许不能很好地对数据建模，还有其他可用于 RBM 隐藏单元和可见单元的选项。以下是对各种情况的数据建模的单元类型。

- 多项式可见单元
- 高斯可见单元
- 二元单元
- ReLU

多项式单元用于主题建模、构建推荐系统，以及使用 RBM 构建分类器（构建分类器时，在顶部抛出一个在无监督的预训练之后用于分类的输出层）。与二元可见单元相比，多项式可见单元用 softmax 函数代替逻辑函数，并使用伯努利隐藏单元。

当处理图像或语音时，二元可见单元给出的特征表示较差。在这种情况下可使用高斯单元，然而隐藏单元会继续使用二元单元，因为对这两种情况使用高斯单元将导致训练结果更不稳定。

在图像和语音应用中应使用高斯单元，因为二元单元在这些场景中往往表现不佳，我们需要比二元单元更低的学习率。在某些情况下，这些单元在学习过程中的表现不稳定。在处理稀疏数据时，我们会使用二元单元。

当处理连续数据时，我们使用 ReLU。ReLU 单元使用的参数数量与二元单元类似，但 ReLU 单元更具表现力。ReLU 单元通常使用比二元单元更低的学习率，以便在训练期间稳定性更好。表 7-5 是根据输入数据查找单元类型的快速索引表。

表7-5：RBM单元类型

数据类型	可视单元类型	隐藏单元类型
文本	高斯（神经词嵌入）、二元（词袋的情况）	ReLU（神经词嵌入）、二元（词袋的情况）
音频 / 时间序列	高斯（连续的情况）、二元（0-1 的情况）	ReLU（连续的情况）、二元（0-1 的情况）
图像 / 视频	高斯（零均值和单位方差的情况）、二元（0-1 的情况）	ReLU（零均值和单位方差的情况）、二元（0-1 的情况）

7.3.3 用RBM正则化

正则化是机器学习中的一个重要课题，因为需要控制参数向量中权重的大小。在 RBM 中使用正则化的一个主要原因是：在训练的早期，有时值很大的权重和相关隐藏单元的值会被“卡”在打开或关闭的状态。正则化有助于训练过程“解开”这些单元。

对于 RBM，需要为二元隐藏单元的活动设置一个稀疏目标。这是对二元隐藏单元有效的期望概率，这个因子通常小于 1.0。

当训练 RBM 时，好的做法是设置 L2 先验权重成本系数和权重衰减在 0.01 到 0.000 01 的范围内。对于 RBM，权重成本通常不应用于隐藏和可见的偏置，因为它们的数量较少，并且过拟合的可能性较小。在某些情况下，我们希望偏置变得更大，因此在这里应用权重

成本就没什么意义了。对于 RBM 的权重成本，一个很好的经验法则是以 0.0001 开始。权重成本的微小差异可能不会改变训练过程。实践表明，Dropout 应用于 RBM 也很高效。

7.4 DBN

DBN 涉及两个阶段的训练过程：预训练和调优。预训练主要涉及从输入数据中学习高级特性，以便更好地初始化前馈网络，为调优阶段提供一个好的起点。DBN 在预训练阶段使用 RBM 集合来学习特征。第 4 章详细介绍了 DBN 和 RBM 之间的相互作用，可回顾第 4 章了解更多细节。

通常使用标准差为 0.01 左右的零均值高斯分布的小的随机值来初始化 DBN 的权重。随着初始值变大，初始学习会更快，但副作用是最终得到的模型可能不好。将隐藏的偏置值设置为 0 通常是一种好的做法。

训练神经网络时，更具体说来，训练 DBN 时，学习率通常在 0.001 到 0.1 的范围内。学习率过高的一个副作用是重建误差和权重通常显著增加。在大多数情况下，这种效应产生的模型较差。



设置学习率可视化

使用 DBN 时，设置学习率的一个好方法是查看权重以及权重更新值的直方图。权重更新值应该是权重值的 10^{-3} 倍。

7.4.1 利用动量

在某些情况下，动量可以使学习过程将参数移动到不是最陡的、正常的方向。这使得学习过程探索搜索空间中某些可能具有相反梯度的相邻区域，而不必在返回之前等待梯度增加的全部效果。它使得随着时间推移，学习方向的变化更平滑，避免了不稳定的振荡。

在预训练阶段，可以调整对比散度的动量来训练 RBM。通常从 0.5 的动量开始设置，在重建误差稳定下来后将其增加到 0.9 左右。如果这对重建误差中造成太大的影响，那么将学习率降低，直到它稳定为止。

在训练开始阶段设置动量时，考虑到随机的初始参数值可能使搜索空间中的起始位置不好，而将值设为 0.5 是一个好的起点。随着时间推移，良好的实现会将动量增加到 0.9。当重建误差上升或接近终止条件时，动量将开始下降到 0.0。

7.4.2 使用正则化

前一节讲解 RBM 的正则化时，讨论了使用正则化（以及将它扩展到预训练 DBN）的主要原因。

可以把 RBM 中的无监督预训练看作正则化的一种形式。无监督预训练相当于对参数空间的一个约束，算法可以在该区域内扫描。事实表明，当有足够的神经元或参数时，无监督的预训练是一个依赖数据的正则化器，然而无监督的预训练也被证明在某些情况下

会损害泛化。这在训练数据集相对较小时容易发生（其中的“小”指少于 100 000 行的数据）。

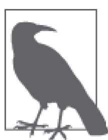


稀疏性与 DBN

有时查看不活跃的特征是提高模型表现的好方法。可以设置稀疏目标，即二元隐藏单元活动的概率（例如稀疏设置远小于 1.0）。研究表明，好的 RBM 稀疏目标值在 0.01 到 0.1 之间。这个因子的衰减率应设置在 0.9 到 0.99 之间。

Dropout

当使用 Dropout 进行预处理时，可使用没有权重约束的较低的学习率来避免预处理技术所发现的特征检测器丢失。Dropout 在 DBN 的调优阶段也是有用的。



MNIST 和 Dropout

在一个已公开的对 MNIST 数据集使用 Dropout 的实践中，隐藏层中 50% 的 Dropout 设置比不使用 Dropout 的表现更好。对输入单元执行额外 20% 的 Dropout 能够进一步提高表现。输入单元上 20% 和隐藏单元上 50% 的 Dropout 设置常被证明对于数据集是最优的。

7.4.3　确定隐藏单元的数量

在考虑 DBN 隐藏单元的数量时，面临的主要问题是描述每个输入训练向量需要多少位（才能产生一个好的模型）。

在这种场景中，面临的主要问题是过拟合。之前讨论过控制方法了，为了确定之后的层中隐藏单元的数量，建议在之前数量的基础上使参数数量小一个数量级。

第8章 向量化

纽约，老圣乔，阿尔伯克基，新墨西哥，
这台旧钻机嗡嗡作响，她做得很好。
如果有人想知道这是怎么一回事，
告诉他们我在寻找那条长长的白线的尽头。

——美国乡村歌手 Sturgill Simpson，歌曲 *Long White Line*

8.1 机器学习中的向量化方法

本章旨在介绍如何将机器学习领域中使用的各种数据向量化。你可能疑惑为什么一本关于深度学习的书会介绍向量化，主要原因是大多数机器学习图书只关注算法本身，而较少关注数据挖掘的完整生命周期。我们希望用机器学习工具尽可能快地试验数据，并且不会花费太多时间在文本数据的自定义向量化等方面。

在与企业客户合作的过程中，我们有过这样的经历：所探讨的是文本分类技术的实现，但是由于长时间讨论将文本转换为向量的基本原则而导致工作偏离轨道。公司有很多简单的数据源，比如可以导出为 CSV 格式的电子表格，但是这样的数据仍然需要转换为向量。我们还试图向客户解释文本数据能以无数种方式被向量化。根据涉及的工具和期望的分类算法的不同，客户可能没有那种与实际统计建模本身无关的广博的编程经验，也没有尝试将文本向量化的好方法。



原始数据与自动特征学习

第 3 章探讨过，深度学习的一个重要特点是从特征工程转向自动特征学习。自动特征学习是深度学习的一个有用特性，但是仍然需要将原始数据转换成工具可以操作的形式，原始数据向量化技术能够做到这一点。

多年以来，一个显而易见的事实是，虽然向量化技术（连同数据处理过程本身）是数据科学过程的核心，却被多次无情忽略。在创作本书时，我们有一种很强烈的使命感：以一种既能完成深度学习建模过程，又不因那些笨拙的创建向量的辅助编程练习而使你分心的方式，为你的向量化处理打下一个坚实的基础。另一方面，在本书中随着从理论转向实践，我们会过渡到：重点在以易于理解的方式处理向量，直接为深度学习模型提供大量高质量的数据。希望你尽快参与对数据的建模，弥补这个差距是解决此问题的一个很好的办法。

使用标准的机器学习数据集是显示其不便性的一个很好的例子。机器学习的向量化阶段可能持续数小时到数天，这会影响你对编程和向量化的感受。这个阻抗因素往往会使许多新用户对开始进行统计模型的工作望而却步。

8.1.1 为什么数据需要向量化

在机器学习和数据科学的工作过程中，需要分析所有类型的数据。一个关键的需求是能够获取每个数据类型并将其表示为数值向量（或者在某些情况下表示为多维数组）。神经网络仍然需要将输入数据表示为向量和矩阵，因为它不能直接在文本、图形和其他非向量/矩阵形式的数据上工作。

处理向量化有许多不同的方法，不同的预处理步骤给输出模型带来的效率也不同。通常建模练习取决于输入数据向量化的好坏程度。输入数据可以有多种形式，包括：

- 列式 CSV 数据
- 文本文档
- 图像数据
- 音频数据
- 视频数据
- 顺序数据

下面的输入数据是加州大学尔湾分校维护的原始鸢尾花数据集中的列式 CSV 数据的样本。

```
5.1,3.5,1.4,0.2,Iris-setosa
4.9,3.0,1.4,0.2,Iris-setosa
4.7,3.2,1.3,0.2,Iris-setosa
7.0,3.2,4.7,1.4,Iris-versicolor
6.4,3.2,4.5,1.5,Iris-versicolor
6.9,3.1,4.9,1.5,Iris-versicolor
5.5,2.3,4.0,1.3,Iris-versicolor
6.5,2.8,4.6,1.5,Iris-versicolor
6.3,3.3,6.0,2.5,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
7.1,3.0,5.9,2.1,Iris-virginica
```

儿童绘本书 *Go Dogs, Go!* 中的一段话可以作为文本文档的一个例子。

```
Go, Dogs. Go!
Go on skates
or go by bike.
```

这两种情况都涉及不同类型的原始数据，但是都需要某种程度的向量化才能成为机器学习

需要的形式。机器学习算法的输入数据需要是连续的稀疏向量格式，如下所示：

```
1.0 1:0.7500000000000001 2:0.41666666666666663 3:0.702127659574468
  4:0.5652173913043479
2.0 1:0.6666666666666666 2:0.5 3:0.9148936170212765 4:0.6956521739130436
2.0 1:0.45833333333333326 2:0.3333333333333336 3:0.8085106382978723
  4:0.7391304347826088
0.0 1:0.1666666666666665 2:1.0 3:0.021276595744680823
2.0 1:1.0 2:0.5833333333333334 3:0.9787234042553192 4:0.8260869565217392
1.0 1:0.33333333333333333 3:0.574468085106383 4:0.47826086956521746
1.0 1:0.7083333333333336 2:0.7500000000000002 3:0.6808510638297872
  4:0.5652173913043479
1.0 1:0.916666666666667 2:0.6666666666666667 3:0.7659574468085107
  4:0.5652173913043479
0.0 1:0.08333333333333343 2:0.5833333333333334 3:0.021276595744680823
2.0 1:0.6666666666666666 2:0.8333333333333333 3:1.0 4:1.0
1.0 1:0.9583333333333335 2:0.7500000000000002 3:0.723404255319149
  4:0.5217391304347826
0.0 2:0.7500000000000002
```

将原始数据转换成适合机器学习的向量的过程分为两个阶段：

(1) 向量化
(2) 规范化

稍后会介绍这两个阶段。我们将在深度学习的以下三个关键领域中研究数据向量化和规范化的概念。

- 序列数据
- 图像数据
- 文本数据

特别指出这些数据类型，是因为它们与深度学习的关系。本书介绍过 RNN 善于处理序列数据，这说明关键的序列向量化技术是有价值的。深度学习在 CNN 的图像分析领域中也显示了它的价值，所以我们会研究图像向量化处理的子主题。本章最后总结文本向量化方法，介绍如何使用 Word2Vec 进行向量化。



深度学习与数据准备的变化

实践证明，深度学习可以减少这些特定的特征工程的预处理步骤。数据仍然需要转换成向量形式，但是深度学习在学习数据集结构的过程中擅长执行特征选择和降维。

过去十年中，许多机器学习实践者倾向于使用专家领域知识手工处理输入向量数据集的特征。专家领域知识指的是对数据如何生成，或数据源如何与现实世界互操作来对现实生活中产生特定影响的深入理解。

本章稍后将介绍，属性较少的数据集是手工特征工程的理想数据，但是较大的数据集，如文本文档、图像或音频文件等，需要用算法来解决。在其他情况下，常见的向量化技术可以自动生成向量，这主要是由于源数据（大量文本）的复杂性使任务变得困难。算法向量

化的例子包括核哈希、词频－逆文本频率（TF-IDF）和 Word2Vec。

下面是从原始数据建立模型时需要考虑的一些主要事项。

- 我们处理的是怎样的源数据？
- 我们想用这个数据训练怎样的模型？
- 我们将数据向量化的方法是什么？
  - 我们是要对这些特征采用手工编码还是使用算法？
  - 原始文本处理起来比较麻烦，那么我们该如何应对？

源数据类型不同，在向量化过程中需要考虑的事情也不同。例如很多人都知道列式数据（如数据库表）向量化的做法，那就是将每个列的类型向量化。然而，如果将散布在多个文件中的多变量的时间序列数据向量化，该怎么做呢？在如今的应用中，这些类型的问题都隐藏在机器学习流水线架构的身后，本章将讨论构建这些流水线的现实问题，首先回顾一下如何处理列式数据的向量化。

抽取、转换和加载

在当今的企业中，许多商业智能和报表工具都具有被称为“抽取”（extract）、“转换”（transform）和“加载”（load）（即 ETL）的预处理阶段。在 Apache Hadoop 中，这也是常见的术语。在这个阶段，通常将多个数据集合并，过滤掉不需要的列，应用必要的转换，然后将数据加载到一个位置供另一个应用程序使用。大部分向量化过程在流水线的 ETL 阶段执行，并且需要考虑构建流水线时应该执行的操作。

8.1.2 处理列式原始数据属性的策略

列式源数据的形状和大小各异，可以把数据表中的每一列看作数据的“属性”，例如从关系数据库管理系统（RDBMS）导出的表。数据集的每个属性可以被进一步分类。因为是工作在数据科学家的领域，所以我们使用他们对数据列或属性的定义。统计教科书通常将数据属性分为以下四类：

- 名目
- 顺序
- 等距
- 等比

下面快速了解每一类。

1. 名目

名目属性有时被称为**枚举的**、**分类的**或**离散的**（例如“晴天”“阴天”和“下雨”）。术语“分类”指这些属性被设置在一组有限的可能性中。它们的符号值不同，这些值用作标签。术语“名目”源于拉丁语的“名称”（name）一词。名目属性或标签彼此之间没有关系，也不隐含任何顺序。

对于名目列，推荐使用列值的独热表示。这将在特征向量中创建多个列条目，其中包含源

数据的列值为 1.0，其他列值为 0.0。

表 8-1 展示了两个记录的例子，其中第一条记录的值为“晴天”，第二条记录的值为“下雨”。

表8-1：特征向量中独热表示的示例

[其他列]	晴天	阴天	下雨	[其他列]
...	1.0	0.0	0.0	...
...	0.0	0.0	1.0	...

2. 顺序

顺序值除了有顺序这点之外，其他方面就像名目值。顺序值有顺序，这样就引入了排序的概念，但没有值之间距离的概念。顺序值可以比较，但是数学运算在这些值的语境中没有意义。

顺序值的例子有“热”“温暖”和“冷”（只要约定的顺序一致即可，排序的方向不重要）。顺序值的另一个例子是“低”“中”和“高”。这些值最终被转换为整数值（例如冷 = 0、温暖 = 1 等），但是在向量化的代码级别上使用浮点值表示。

顺序和名目之间只有细微的差别。一些旧的数据挖掘系统倾向于只使用名目和顺序类型。如果为输出向量选择向量表示方法，我们会使用独热表示。对于输入向量，我们可能将列值转换为实值。

3. 等距

等距值按固定且相等的单元来排序和测量，例如特定的日期或年份。等距可以比较，但是等距的相加和相减没有意义。等距值已经是数字了，所以不需要转换，但是最后我们可能使用规范化的方法。

4. 等比

等比值基于零点来测量，并被视为实数。在这种情况下，数学运算是有意义的。等比数据的方案是定义一个零点，从这个固定零点起计算距离。等比值已经是数字了，所以不需要转换。不过我们可能使用一个规范化的方法，稍后将介绍做法。

8.1.3 特征工程与规范化技术

向量化已经存在一段时间了，并且实践者已经开发出了用于构建规范的“扁平”向量变体的模式。这些向量化模式经常应用于机器学习流水线的逻辑回归、随机森林等方法。

一种经典的向量化方法是，创建一个大小固定的长度为 n 的向量（n–1 是特征数量，最后一位是标签值专用位），然后根据设计表示数据的一些启发式规则来设置每个索引单元的值。这通常适用于多层感知器（对那些不使用小批量的）网络。不过在深度学习中，很多情况下深度学习的输入需要构建更复杂的 n 维矩阵数据结构。随着内容的推进，本章将展示为 RNN 创建输入张量和为 CNN 创建四维张量输入的更先进的方法。根据本节的主题，我们将重点介绍如何设计这些输入张量中的单个特征值，以及如何对这些值应用规范化方法。

向量化的过程基于选择属性以及在特征向量中找到为其分配特征的维度。具体做法取决于是否使用 CSV、文本、图像或时间序列数据。我们仍然需要选择输入数据中想处理的那些部分，并在输入向量中收集它们。可以通过多种方式将 n 个属性转换成输出向量中的 m 个特征，这被称为**特征工程**。原始源数据通常需要在建模之前进行转换，其呈现形式如下：

- 原始文本，如文本文件中的文档；
- 每行包含一条推特文本的文件；
- 自定义文件格式的二进制时间序列数据；
- 包含数值和字符串混合属性的预处理数据；
- 图像文件；
- 音频文件。

根据原始数据的条件和来源，属性可以是数值或字符串形式的。大多数情况下，属性的值是数值类型且连续的。

这些属性测量数值包括实数或整数值。在这个设置中，“连续”这个词在意义上指“混在一起的”。

数据清洗与 ETL

对大多数数据科学家来说，数据是需要清洗的，而且会耗费大量时间。数据清洗通常发生在机器学习的 ETL 阶段，并且通常使用脚本或 Hadoop 作业在 ETL 流水线中完成。根据属性的类型和场景的特定语义，可能需要使用占位值。即使数据集中每一列都有一个值，但是也不能完全免除对数据的进一步检查。由于各种原因，仍会出现不准确的值。很多时候，数据是在一段时间内收集的，一些列的值不相关，并且存在着接收到一个不准确的无效值的潜在风险。再次强调，研究数据集和图形属性的统计规则是最佳实践。

特征工程技术可以用于将列式数据手工向量化，或者处理时间序列或图像数据等更复杂的数据。这些技术包括：

- 直接取属性不变的值；
- 规范化属性，以创建特征；
- 特征二值化；
- 降维。

下面是这些技术的一些细节。

1. 特征复制

为了从原始输入数据中创建向量，需要选择那些与模型最相关的数据的合适“特征”。特征选择一直是建立成功模型的关键，所产生的向量中的特征数量通常与源数据中的属性数量不相等。很多时候，源数据将与其他数据集结合在一起，然后所得到的数据集的非规范化视图的属性子集将用于导出向量中的最终特征集。

最常见的生成特征的方法是简单复制一个已经是数字且其在正确范围内的属性，可惜这种情况并不常见。更多的时候，属性需要做一些基本的转换。

> **缺失值的处理**
>
> 使用数据时，原始输入数据中经常出现缺失值，这些值常常由范围外的数据条目表示（可能是数字条目中的 -1，或是永远不能为零值的数字属性的 0）。对于名目属性，缺失值通表示为空白或破折号。缺失值可能由多种原因导致，理解源数据的机制有助于分析出值缺失的原因。一个好的实践是在开始通过查找离群值来构建向量之前研究数据集的统计和属性。一个简单的做法是绘制各种属性的图表，并查找看起来不正确的东西，例如值为 0 的年份属性。
>
> 这里有一些处理缺失值的基本方法。
>
> - 过滤掉缺失值的记录。（警告：如果值不是随机缺失的，这样做可能会引入偏差！）
> - 将缺失值设置为 0。（警告：有时有效；会在数据中引入“噪声”。）
> - 设置为该列的最常用值（有其他确定该替换值的方法）。

2. 规范化

规范化方法在将原始数据转换为向量表示之后，将输入训练数据缩放至一个范围，例如 [0, 1]、[−1, 1] 等。之所以关注规范化，因为它会影响神经网络中的激活值。如果网络的输入太大，会导致激活值太大，可能对训练产生不利影响。相反，如果输入太小，激活值可能也会变得太小。计算网络中的梯度时可能受到类似的影响。



一些超参数假设数据已经规范化

做出权重初始化方案（如 Xavier 权重初始化）等设计决策，是以假设已对输入数据应用规范化为前提的。

在规范化时，需要转换数据，使其更加一致。有两种基本工具能够以不同方式的组合来执行规范化。

- 居中
- 缩放

这两种方法的目标都是使训练过程更容易。居中方法转换特征，使其以原点为中心，或者在大多数情况下，以平均值为中心。缩放则重新调节特征，使特征在整个数据集上的方差为 1，或者使每个特征在训练数据集上的最大绝对值为 1。

运用上述核心技术的具体规范化方法如下：

- 标准化
- 极值缩放
- 白化
- 主成分分析（PCA）

在通常的规范化实践中，应用居中和缩放将数据转换为具有零均值和单位方差。极值缩放被视作基本缩放技术的变体，有助于学习算法更高效地操作。稍后将简要介绍白化和主成分分析。



规范化的缺陷

实践中许多数据集都有离群值。当集规范化数据时，非离群值数据被缩放到一个很小的区间，你应该对此保持警惕。

快速复习：平均值与方差

向量的转换中通常会使用统计度量，其中包括使用样本平均值（如普通的“平均值”）或特征的样本方差（或普通的“方差”）。

特征（或数据集中的列）的平均值只是特定特征值的平均值。对于 N 个实值数 $x_1, x_2, \cdots, x_N$，平均值计算方法如下：

$$\mu = \frac{1}{N}\sum_n x_n$$

特征的方差用于测量特征的值围绕其平均值的变化量。

$$\sigma^2 = \frac{1}{N-1}\sum_n (x_n - \mu)^2$$

其中 μ 是刚才定义的样本平均值。

标准化和零均值，单位方差。当将特征列值“标准化”[1] 时，我们减去位置的度量值（最小值、最大值、中位数等），然后除以缩放的度量值（方差、标准差、范围等）。需要重新调整特征，使它们具有以下标准正态分布的特性。

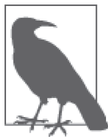
$\mu = 0$（平均值等于0）

$\sigma = 1$（标准差等于1）

对于每个特征，要将值转换为零均值和平均值为 1 的标准差的分布值。要做到这一点，首先需要计算每个特征的平均值和标准差，然后从每个特征中减去平均值，并将结果除以特征的标准差，如下所示：

$$z = \frac{x - \mu}{\sigma}$$

这也称为**零均值，单位方差**。该方程将特征值集中在平均值为 0、标准差为 1 的 [-1, 1] 范围内。



零均值，单位方差是最常用的标准化方法。在神经网络中，常与密集向量一起使用。

注 1：在一些文献中，“标准化”也称“z-score 规范化”。

SGD 等优化方法，会大大受益于标准化等方法。很多时候，当遇到不同数量级的特征时，一些参数会更新得较快，这是因为输入特征在不同的数量级上。标准化也是一种有助于基于距离来比较特征之间相似性的方法。

当向量中数据字段从属性转换为特征时，还需要考虑向量的稀疏程度（即向量中有许多零点）。当数据稀疏时，需要缩放特征（或前面提到的“规范化”）到 (0, 1) 范围内以表示概率。反之，当向量密集时，需要先执行预处理步骤，使平均值为 0、单位方差为 1，然后完成缩放结果值的步骤（现在范围为 [−1, 1]）。

稀疏和密集数据预处理的主要区别是：对于密集数据，首先进行零单元 / 平均运算，然后进行缩放；而对于稀疏数据，我们只进行缩放。显然，对于算法需要数据在 (0, 1) 范围内的情况，应将数据缩放到该范围（如 RBM）。



标准化的注意事项

随意将标准化应用于稀疏向量不是一个好主意。极值标准化（稍后介绍）在某些情况下可能更好。对于稀疏向量，我们通常希望规范化后 0 依然是 0，而标准化可能不是这样做的。



标准化更广泛的表示

[−1, 1] 范围内的数据表示比基本规范化 [0, 1] 的数据表示更广泛，这是因为在浮点表示中前者有更多表示信息的位。

极值缩放。另一种将数据规范化的方法是使用极值缩放，做法是将每个特征缩放到一个固定的范围（例如常见的范围是 [0, 1]）。最基本的极值缩放的方程如下所示：

$$X_{\text{norm}} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}$$

与标准化相比，极值缩放方法要做出的妥协是最终得到的标准差会较小。极值规范化对离群值更敏感，因为单个离群值（小的或大的）会影响极值，并且对规范化有很大影响。标准化方法则不大有这个问题。

极值缩放在图像处理中趋于流行，对于图像处理，像素强度需要标准化到特定范围（例如 RGB 值的范围是 [0, 255]，并且需要将输入向量缩放到 [0.0, 1.0]）。



规范化中冲突的术语

有些文献将极值缩放简单地称为“规范化”，而将“标准化”当作另一种技术。本书把它们都视为规范化方法，并分别具体地表述为“极值缩放”和“标准化”，以进一步区分这些方法。

白化和主成分分析。在深度学习的场景中，有一种需要了解的转换是统计白化（简称白化），这是转换数据使其具有特性协方差矩阵的过程。白化还可以去除数据中的相关性，从而找到一种以高效的方式来表示数据的最佳方法。白化转换的名称来自将输入数据变为白噪声向量的操作。使用白化的一个已知问题是，在某些情况下它会放大数据的噪声。

主成分分析是将数据集中的（可能相关的）数据转换为线性不相关变量的数据集的方法。该方法借助正交转换来执行转换及产生一组主成分。

> **降维**
>
> 如果想以某种算法找到与模型最相关的属性，在机器学习文献中的常见解决方案是降维。一个关于深度学习（具体指 DBN）有趣的动态是，预训练阶段通过训练 RBM 的层来学习数据的表示。自动编码器将原始数据编码成较短的数字向量。这个较小的表示之后可以作为输入，用于分类或相似性搜索算法。

在 RNN 和 CNN 中应用规范化。当模型假设输入为某种结构时，数据也应该规范化。当使用 RNN 的变体计算给定输入的平均值、标准差、最小值或最大值时应考虑所有时间步，如 LSTM。

当在 CNN 的场景进行规范化时，我们希望在图像中的所有像素位置上共享通常是针对每个通道（RGB）分别计算的平均值、标准差和其他计算值。

回归模型的规范化。虽然需要在回归的场景中规范数据，但是需要把回归当作一种特殊情况。规范分类模型时，通常是规范特征（输入），然而对于回归，还经常需要规范标签（例如目标或输出数据值）。

对于回归模型，在规范化过程中应使用相同的基本做法。当规范训练过程中的输入和输出时，仍然使用极值缩放和标准化之类的方法。

> **为什么要规范回归的输出**
>
> 考虑一下均方误差（MSE）损失函数的例子：如果值的范围是 0~100 万，那么最终会产生大的误差和过大的梯度。
>
> 不过在规范标签（输出值）之后，将无法预测所关心的原始数据，因此需要撤销对网络预测的规范。可以执行极值和标准化的反向规范，因为它们是一对一映射的。
>
> 重新排列规范化方程，即可得到撤销它的方程。
>
> （反向标准化）
> origScaleOutput = netOutput × sigma + mu
> （反向极值缩放）
> origScaleOutput = netOutput × (xMax − xMin) + xMin

3. 二值化

当从属性构建特征时，存在其他情况，建模算法可能要求数据遵循多变量伯努利分布。

在这种情况下，可以使用**特征二值化**方法，通过设置数值特征的阈值来获得布尔值。这涉及使用过滤器产生一个特征，这个特征以 1 或 0 为值。

8.2 使用DataVec进行ETL和向量化

本章介绍的数据 ETL、向量化和规范化等对实践者来说都是重要的主题。我们不仅要考虑建立什么样的机器学习流水线，还要考虑如何操作这个流水线。考虑到大多数连接和 ETL 操作是数据集上的单次操作，因而当我们希望流水线操作更快时，水平扩展也会发挥作用。



并不是只有在数据量很大时才需要更快的速度

即使你没有 TB 级的输入数据，也可能需要考虑机器学习流水线在 ETL 阶段可扩展的解决方案。很多时候，管理者或客户会问："我们怎样才能更快地得到答案？"大部分情况下，答案都是水平扩展流水线。Apache Hadoop、Apache Spark、DL4J 和 DataVec 等工具都是现代可扩展机器学习流水线解决方案。

DataVec 允许以本地模式执行，也可以在 Apache Spark 和 Apache Hadoop 上水平扩展。在第 5 章的示例中，可以看到直接使用 ND4J API 来操作数据以生成供 DL4J 库使用的输入 `DataSet` 对象。DataVec 的长处之一，是它能够接受某些类型的标准数据并自动生成向量化的 `DataSet` 对象。DataVec 还包括转换以收集向量化数据的统计数据，然后将 `DataSet` 对象中的数据规范化。

如本章之前所讨论的，机器学习（和深度学习）中对向量化的基本需求是能够处理列式（CSV）数据类型。在 DataVec 本地模式下，只需要编写以下代码即可：

```
RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );

//创建DataSetIterator。这里我们假设分类的情况：

int minibatchSize = 10;    //每个小批量中的样本数

int labelIndex = 7;        //包含标签在内的列的索引
int numClasses = 5;        //类别数（标签类别）
DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

最早在第 5 章中用 CSV 数据输入建立多层感知器模型时出现过这段代码。DataVec 允许我们指定不同类型的记录读取器，将它们与 `DataSet` 迭代器结合使用，并在 DL4J 建模流水线中处理数据。DataVec 能够让我们快速指定下列操作：

- 连接
- 过滤器
- 规范化
- 标准化

随着本章内容的推进，将讨论其他主要的数据类型，并展示 DataVec 是如何处理它们的。



如果想阅读更多关于 DataVec 的内容，请参考由特邀作者 Alex Black 编写的附录 F。

8.3 将图像数据向量化

图像是挖掘信息的另一个丰富的来源。图像的定义为“描绘或记录视觉感知的人造物，例如二维图像”。在表 8-2 中展示的像素值的数组表示存储在计算机上的图像。每个像素都有一个值，指示像素的亮度或像素的颜色。

表8-2：以行、列形式可视化图像中的像素

	列1	列2	...	列*m*
行 1	p11	p21	...	p-*m*-1
行 2	p12	p22	...	p-*m*-2
...	...	...	...	...
行 *n*	p-1-*n*	p-2-*n*	...	p-*m*-*n*

每个像素的位数决定它可以显示多少种颜色。1 位像素只能显示二值图像（通常是黑白像素）。八位像素更常见，每个像素可以显示 256 种颜色（或灰度）。对于灰度图像，像素的整数值表示像素的亮度（范围为从黑色到白色，或者 0 或 255）。要表示彩色图像，必须为每个像素指定单独的红色、绿色和蓝色的量（假设为 RGB 颜色空间），因此像素值实际上是三个数字的元组。常见的图像文件格式有 JPG、PNG 和 GIF。



图像格式

每种格式以不同的方式存储图像数据，并对原始数据执行不同级别的压缩。图像不同于文本，能够更直接地向量化。



处理视频数据

视频数据向量化是图像向量化和时间序列向量化的变体。在视频数据中有一系列带时间戳的图像。视频向量化的过程涉及随时间跟踪图像向量化，并且从要考虑的单个向量（例如单个帧与跨多个帧的图像的子集）的角度来看，这个过程可能更复杂。

8.3.1 DL4J中的图像数据表示

在深度学习中，我们最感兴趣的是 CNN 的图像向量化。在 DL4J 中，每个图像转换为 `INDArray` 对象中的三维张量表示（有关该主题的更多信息，请参阅附录 E），如图 8-1 所示。

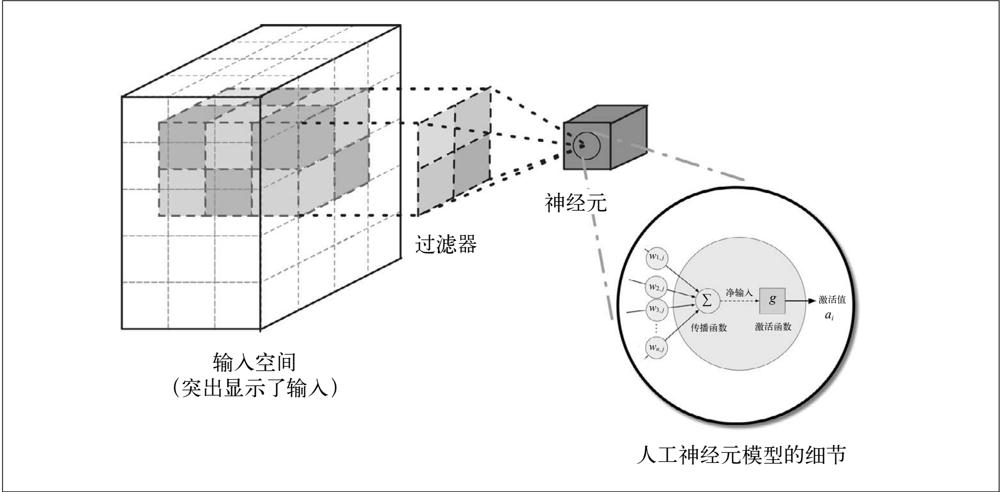


图 8-1：卷积网络中的三维输入空间

从本书的示例可以看出，当创建用于训练的图像小批量时，实际上正在创建具有以下维度的四维（4D）张量。

(1) 小批量大小
(2) 深度
(3) 高度
(4) 宽度

其中，宽度和高度直接映射为输入图像的特征，深度是图像数据中通道的数量（例如 RGB 图像中通道的数量通常为 3）。



4D 小批量图像是 CNN 的标准

4D 格式是在 DL4J 中使用 CNN 处理图像的标准。

如果要为多层感知器网络将数据向量化，则需要将数据扁平化到二维（例如 `minibatchSize`，深度 × 高度 × 宽度）。为了理解这种图像向量化的变体，需要继续对视觉网格的解释。对于图像，首先要做的是从像素数组中的每个位置提取像素强度。它的基本思想是采用 $m \times n$ 图像的概念，并将这个矩形扁平化为 $1 \times (m \times n)$ 数组，如表 8-3 所示。

表8-3：扁平化后的图像数据

	列1	列2	...	列*m*-x-*n*
行 1	像素 1	像素 2	...	像素 *m*-x-*n*

图像由一系列浮点值组成，这些值在物理上表示色彩强度，但是我们往往将其看作像素强度的 $m \times n$ 网格的逻辑形式。为了使图像适合任何线性代数向量，需要将这些像素强度分配到更适合机器学习算法的值向量中。

如表 8-4 所示，需要将这个矩形转换成矩阵中每个图像记录的 $1 \times (n \times m)$ 数组。使用 `CnnToFeedForwardPreprocessor` 来执行这个扁平化过程，这由在网络配置中使用的 `.setInputType(InputType.convolutional(height,width,depth))` 自动完成。`setInputType()` 方法会意识到我们正在试图将一个 4D 数组输入到一个需要 2D 数组的密集层中，因此它将添加预处理器来将数据扁平化。

表8-4：2D数组中的扁平化后的小批量图像

	列1	列2	...	列*m*-x-*n*
图像 1	像素 1	像素 2	...	像素 *m*-x-*n*
图像 2	像素 1	像素 2	...	像素 *m*-x-*n*
图像 3	像素 1	像素 2	...	像素 *m*-x-*n*

8.3.2　使用DataVec将图像数据与向量规范化

在图像建模中，向量化的一个主要部分是从文件格式容器中获取原始像素数据，并将其转换为学习算法所能理解的向量对象。需要逐个单元读取原始像素数据，进行所需的转换或标准化，然后在输出向量对应的索引处添加值。很多时候，我们只是简单地取整组图像像素，然后将它们直接转换为一个向量，并转换每一个像素。在更先进的图像处理方法中，图像的一部分被解析出来并被自己用作向量。

DataVec 的 `ImageRecordReader` 是图像数据的记录读取器，它还能执行裁剪等常用的图像操作。`ImageRecordReader` 支持多种图像格式，其中包括：

- JPG
- GIF
- PNG
- TIFF
- BMP

JavaCV 和 OpenCV 提供了高效的加载和图像操作。下面的代码片段显示了设置 DataVec 的 `ImageRecordReader` 来构建 `DataSetIterator`，它使得 DL4J 能够直接训练图像数据。

```
ImageRecordReader reader =
    new ImageRecordReader(
        outputHeight, outputWidth, inputNumChannels, labelMaker );
reader.initialize( inputSplit );

//最后，创建DataSetIterator：

int minibatchSize = 10; //每个小批量中的样本数
int labelIndex = 1; //总是将ImageRecordReader设置为1
int numClasses = 3; //类别数（标签类别）

DataSetIterator iterator =
    new RecordReaderDataSetIterator(
        reader, minibatchSize, labelIndex, numClasses );
```

这样的功能使得 DL4J 更容易使用，因为你可以关注整个 ETL 和向量化流水线，而不必操

心细节，比如从文件格式中提取像素值等。附录 F 详细介绍了 `ImageRecordReader`，以及图像数据转换、规范化和标准化的方法。

8.4 将序列数据向量化

除了行的给定列中可能有多个值之外，序列数据类似于列式数据。例如随着时间推移，在特定位置周期性读取温度的温度计。如果给每个温度测量值增加一个时间戳，这个序列数据就可以称作“时间序列数据”。通常将时间序列数据定义为在连续的时间间隔上测量的数据点序列。金融领域的纽约证券交易所每天不同的股票收盘价就是时间序列数据。另一个例子是智能电网上的相量测量单元（PMU）装置的传感器读数，该装置每秒测量 30 次“相位角”和电压。

最常见的一个应用是日志处理。Web 服务器、手机和信用卡刷卡机等系统每次响应请求执行操作时都会生成日志条目。这些设备是序列乃至时间序列数据的巨大来源。传统的 RDBMS 系统不太适合存储这样的数据，所以它们经常被存储在存储区域网络（SAN）驱动器或 Apache HDFS 中。人们谈论最多的物联网（IoT）主要由这些用例组成，其中我们提取和处理来自传感器（Web 服务器、PMU、手机等）的时间序列数据。我们可能不会直接想到的序列数据的其他来源是基因组数据和规则字符序列（如句子）。

现在许多机器学习工具对序列数据的支持很差，实践者最终不得不手动编码复杂的 ETL 流水线来处理这类数据。噪声很多的序列值需要规范化和标准化。本节的剩余部分将研究一些使用 DataVec 处理序列数据的 ETL 和向量化需求的实用方法。

8.4.1 序列数据源的主要变体

基于由谁或什么设备产生数据、在何处存储数据，以及使用什么格式来存储数据，序列数据可呈现多种形式，常见的三种主要形式如下：

- 原始日志；
- 时间序列数据的单个 CSV 文件；
- 时间序列数据的多个 CSV 文件。

这些源文件可能有不同的数据排列方式。有了实体（例如“用户”或跟踪动作的其他代理），以及为每个序列条目收集的 *N* 个值，我们就拥有了一个序列，其中每个实体都有一个或更多列。数据在磁盘上存储的形式包括：

- ❑ 单个文件中包含所有序列

  每一行是读取自单个源的 CSV 集合（例如多列）。

- ❑ 跨多个文件的序列扩展

  其每个文件通常都用来表示读取自单个源的信息。

在单个文件中，有时会看到在 CSV 文件中以行格式排列的序列数据，其中每一行代表给定源的单个列。该行还包含源的标识符。

当每个源在每个时间步产生了多列时间序列，就会看到数据以每行即一个时间步的形式出

现在 CSV 文件中。每个文件都代表多变量时间序列数据的一个单个源。每行通常有一列或多列，表示源中每个时间步的多个值。

还需要考虑在时间序列数据的每个时间步中包含多少列的测量值。例如当只测量电网中的电压时，在一个序列的每个时间步中，我们只会测量一列的值。如果每一步（或者时间步）都要测量电压和温度两个值，就需要一种能够每一步表示至少两列的文件格式。

所有的这些（以及更多的）因素在 ETL 过程中起作用，最终导致序列数据的向量化、规范化和标准化。最终所有这些信息都被放到一个对象（例如 `DataSet` 对象）中，以便建模工具能够理解。DL4J 通过 DataVec 为处理序列数据 ETL 提供支持。

8.4.2 使用DataVec将序列数据向量化

我们需要能够从驱动器（本地或从另一个位置，如 HDFS）加载原始序列文件，这涉及以下任务：

- 读取文件格式；
- 处理数据的连接或转换；
- 匹配标签（如有）与序列相关源的特征；
- 执行所需的规范化或标准化；
- 最后生成数据正确对齐的 `DataSet` 对象。

理想情况下，输入文件格式与在 DataVec（或 Spark，如果使用 Spark 执行环境）中已有的记录读取器匹配。最后用 ND4J 中由 NDArray 表示的张量结构中的一列或多列构造一个小批量序列数据。我们想生成一个 3D 张量结构，如图 8-2 中的右图所示。

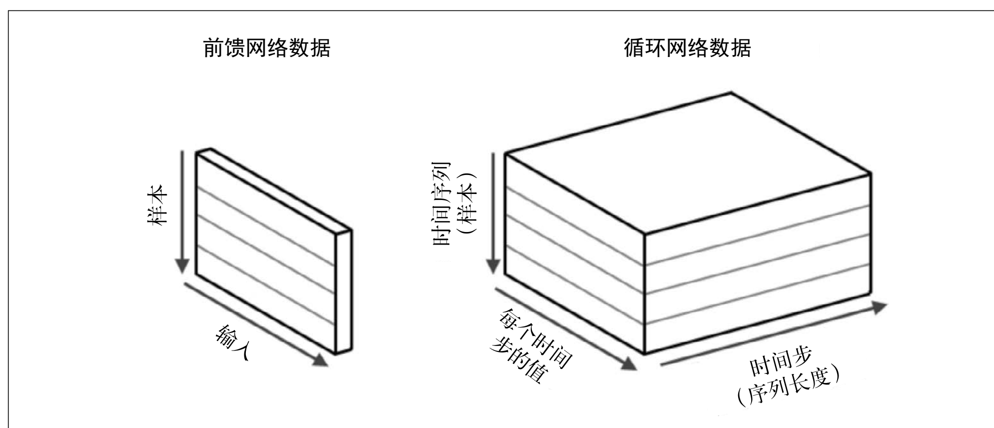


图 8-2：RNN 的序列数据表示

创建一个满足以下条件的 3D 张量（例如 3D 矩阵）。

- 小批量大小是第一维度。
- 特征列是第二维度。
- 每一列的时间步的值是第三维度。

执行上述所有 ETL 工作可能变得复杂，需要人工对齐 NDArray 张量中的数据值。从这个角度，建议在可能的情况下，使用 DataVec 中的记录读取器和格式。



自定义文件类型

如果在 DataVec 中没有用于特定类型输入数据的记录读取器，则必须为构造张量编写一些自定义代码。根据数据的复杂性，这可能是一个非常有挑战的练习。

在继续讨论如何使用 DataVec 将一些序列转换为张量的模式之前，首先了解时间序列向量化的一些较老的变体。

1. 将时间序列转换为单个向量

对于大多数机器学习方法，每条记录需要表示为向量或矩阵中的行。每行表示单条记录，一组行表示要训练的这些向量的小批量，如图 8-2 中左图所示。

这种做法的主要缺点是输入数据的时间维度缺失。在 RNN 出现前的大部分时间里，我们不关心这个问题，因为大多数机器学习技术无法考虑值序列（或时域）。大多数时候，实践者都是发挥各种奇思妙想，人工创建向量。虽然在某些情况下是高效的，但是正如第 5 章所讨论的，这种做法在机器学习流水线的耐用性方面可能导致严重的技术负债。

> **符号聚合近似**
>
> 在平面向量表示的世界中，有几种处理时间序列数据的方法。其中一种处理噪声时间序列数据的方法是在其上运行低通滤波器作为预处理步骤。符号聚合近似（SAX）预处理可以很好地减少时间序列数据中噪声。Eamonn Keogh 博士的团队在加州大学河滨分校开发了 SAX。SAX 是时间序列的符号表示，具有一些特性。SAX 表示 d 维空间中长度为 n 的时间序列 T。它本质上是带有下限的欧几里得距离的低通滤波器。SAX 对原始时间序列数据作降维处理，这有助于学习算法。
>
> SAX 的继承者是可索引的符号聚合近似（iSAX），它增加了一些额外的能力。iSAX 修改了 SAX，以便于“可扩展哈希”和多分辨率表示，并允许快速精确搜索和超快速近似搜索。最终的效果是，iSAX 在表示不同值的分布方面更加灵活，它使得在许多应用中（如快速近似搜索）使用时间序列上的兴趣指数更方便。

2. 在本地模式下将序列数据转换为`DataSet`对象

在 RNN（和 LSTM）中，我们想生成一个由 `DataSet` 对象表示的 3D 张量。为了说明这个概念，来考虑这样一种情况：对数据建模，其中每个实体的序列数据被分成不同的文件，每个实体的标签也有相应的文件，如下面的例子所示。

- 实体 0 的序列数据的文件：
  - train/features/0.csv
- 实体 0 的序列数据的标签：
  - train/labels/0.csv

该数据在每个测量步骤中只有一列数据，所以它被视作单变量。使用 DataVec 时，可以以 `CSVSequenceRecordReader` 为记录读取器，然后，`NumberedFileInputSplit` 类将处理文件和目录命名方案。使用 `SequenceRecordReaderDataSetIterator` 类来生成 `DataSet` 对象，它将使用记录读取器和输入分割对象，如下面的代码示例所示：

```
//请注意，我们有450个表示特征的训练文件：从train/features/0.csv 到 train/
features/449.csv

SequenceRecordReader trainFeatures = new CSVSequenceRecordReader();
trainFeatures.initialize(new NumberedFileInputSplit(featuresDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

SequenceRecordReader trainLabels = new CSVSequenceRecordReader();
trainLabels.initialize(new NumberedFileInputSplit(labelsDirTrain
    .getAbsolutePath() + "/%d.csv", 0, 449));

int miniBatchSize = 10;
int numLabelClasses = 6;

DataSetIterator trainData =
    new SequenceRecordReaderDataSetIterator(trainFeatures,
    trainLabels, miniBatchSize, numLabelClasses,
    false, SequenceRecordReaderDataSetIterator.AlignmentMode.ALIGN_END);

//将训练数据规范化
DataNormalization normalizer = new NormalizerStandardize();
normalizer.fit(trainData); //收集训练数据的统计信息
trainData.reset();

//使用先前收集的统计信息立即进行规范化。
//由"trainData"迭代器返回的每个数据集将被规范化。
trainData.setPreProcessor(normalizer);
```

这里将前面提到的对象汇总起来创建 `DataSetIterator`，它将提供 `DataSet` 对象的小批量。这些 `DataSet` 对象的小批量将包含来自源文件（和标签文件）的正确对齐的数据。在代码的最后，还可以看到在 DataVec 流水线运行时如何将规范化和标准化应用于数据。

以下是 DataVec 记录读取器的其他变体。

- `CSVNLinesSequenceRecordReader`

  `CSVNLinesSequenceRecordReader` 是前面提到的 CSV 序列读取器的另一个版本。每个序列都正好是 N 行长，全部记录在一个文件中，一个接一个。

- `RegexSequenceRecordReader`

  `RegexSequenceRecordReader` 对日志数据很有用，需要正则表达式来解析这些列。

在处理文件格式方面，前面演示了 `NumberedFileInputSplit` 的使用，它知道如何读取在目录中分离的编号文件。显然还有许多其他方法可以在驱动器上布置文件，我们会对 `FileSplit` 类进行子类化以进一步处理这些场景。

这里要考虑的是为那些在 DataVec 中没有所需内置功能的序列数据构建自定义 `DataSet`，以及在 Spark 上运行 DataVec。稍后将介绍 `DataSet` 对象的自定义构造。第 9 章将探讨在

Apache Spark 上将序列数据向量化。

3. 从序列数据构建自定义DataSet

创建在 DL4J 中训练 LSTM 所需的数据结构只需要以下内容：

- 持有训练数据的输入 NDArray；
- 持有标签数据的标签 NDArray；
- 表示掩码信息的两个掩码 DataSet。

将这四个 NDArray 对象组合在一起，构建 DL4J 期望的 DataSet 对象。

```
DataSet d = new DataSet( input, labels, mask_in, mask_labels );
```

实际上，诀窍在于以与前面的行类似的方式，在把所有的数据合并之前，把每个 NDArray 数据放到正确的位置，如下面的代码片段所示：

```
//分配空间：{小批量大小，列数量，时间步数量}
INDArray input  =
    Nd4j.zeros(new int[]{ miniBatchSize, inputColumnCount, maxTimestepLength });
INDArray labels =
    Nd4j.zeros(new int[]{ miniBatchSize, outputColumnCount, maxTimestepLength });
INDArray mask   = Nd4j.zeros(new int[]{ miniBatchSize, maxTimestepLength });

for (int miniBatchIndex = 0; miniBatchIndex < miniBatchSize; miniBatchIndex++) {

  for ( int curTimestep = 0; curTimestep < endTimestep; curTimestep++ ){
    //input ->设置字符id-index的列索引 ->在当前时间步（c）
    input.putScalar(new int[]{ miniBatchIndex, columnIndex, curTimestep }, 1.0);

    //在这里做更多的列输入

    //现在设置掩码，设置所有有数据的时间步为1.0
    mask.putScalar(new int[]{ miniBatchIndex, curTimestep }, 1.0);

    //labels ->设置下一个字符id-index的列索引 ->在当前时间步（c）
    labels.putScalar(new int[]{ miniBatchIndex, nextValue, timestep }, 1.0);

  }

}

INDArray mask2 = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
Nd4j.copy(mask, mask2);
return new DataSet(input,labels, mask, mask2);
```

显然，在执行这段代码的时候，原始数据所需的所有 ETL 和向量化已经执行了，这时只是在张量数据结构中对齐数据。下面是关于这段代码的主要注意事项。

- 需要使用 ND4J 构造输入矩阵。
- 双回路模式通常是以一种连贯的方式穿越 3D 数据结构并建立数据。
- 然后主要调用 INDArray.setScalar() 来设置值。

需要注意排列标签矩阵数据的方式。如果像第 5 章莎士比亚的例子一样做字符预测，下一

个字符值要放在标签矩阵的当前时间步中。如果正在执行序列数据分类模型（例如对日志数据中的异常进行分类），类别可能放在标签矩阵中的每个时间步中。布置数据是一门艺术，处理在不均匀的时间步中发生的数据有多种策略，你需要自行判断采用哪种策略。

在张量中使用掩码

在 DL4J 中，训练数据和标签中都使用掩码。

在训练数据掩码中，每个包含训练数据的时间步的掩码值设为 1.0，其他所有时间步的掩码值设为 0.0。对于输入数据结构和标签数据结构，通常使用相同的掩码。

在代码中实现时，通常遍历张量数据结构，为每个时间步设置掩码条目，代码如下所示：

```
INDArray mask = Nd4j.zeros(new int[]{ miniBatchSize, maxLength });
...
for (...) {
   ...
   mask.putScalar(new int[]{ miniBatchIndex, timeStep }, 1.0);
```

注意，这里只设置了一个小批量索引，然后在掩码数据结构中设置了一个时间步索引。不需要指定哪些列有数据，只要特定的时间步中任意列有数据即可。

8.5　将文本向量化

由于文档或文章包含任意数量的单词，而且语料库中的单词数量也不相同，因此乍看上去文本的处理会很麻烦。如果每个文档没有一个一致的“属性”数量，那么最简单的特征工程方法（简单地复制属性值）将无法工作。我们需要将可变数量的属性转换为一致数量的特征的方法。有很多方法可以做到这一点，稍后介绍。首先介绍一些较早的文本向量化方法，然后将它们与更现代的方法比较，如 Word2Vec。接下来先介绍自由文本的基本原理和向量空间模型（vector space model，VSM）。

VSM 是一种常用的文本文档向量化方法。在这个模型中，每个可能的单词被映射到一个特定的整数。如果有足够大的数组，每个单词都可以映射到数组中唯一的位置，每个索引处的值是单词出现的次数。通常数组小于语料库词汇表，因此需要一个向量化策略来解决这个问题。

为了对文本建模，需要经历以下几个阶段。

(1) 句子分割：可根据情况直接跳转到分词阶段。
(2) 分词：在这个阶段发现单独的词。
(3) 词干提取：在这个阶段找到词的词根或词干（可选）。
(4) 词性还原[2]：去除单词的不同屈折变化，将其恢复为基本词形（可选）。
(5) 移除停止词（可选）。

注 2：在一些语言中，词干提取是词形还原的“轻量级”版本。

(6) 向量化：根据前面的输出，产生浮点值数组。

使用 VSM 时，我们假设单词有维度，并且彼此之间是正交的，这就类似于一个点的 x 和 y 值被视作独立的。但对于文本来说，这并不适用，因为单词之间有相关机制。例如“田纳西志愿者”这样的产品或团队的名称，“田纳西”和“志愿者”一起出现的概率更高，因此这些词并不是真正独立的。对于向量化，可以使用多种策略。在下面的内容中，为了简单起见，文本向量化空间中最常用的几种方法按照难易程度排列，从最简单的到更高级的。

8.5.1 词袋

词袋是单词的列表和每个词的数量。它是最简单的向量模型，但由于涉及单词数量，它需要使用很多列。通常将每个文档中单词的数量规范化，这将有助于学习算法，使我们能够从文档的向量表示中得到单词出现的概率。词袋模型提供了一个简化的表示，通常用于自然语言处理（NLP）和信息检索（IR）。

一组词或一个文件表示为词袋，或词的多重集，语法和词序被忽略，但是我们仍然跟踪文档中单词出现的次数。词袋向量化技术在文档分类和 IR 领域中应用最为广泛[3]。

在表 8-5 中，向量对文档中每个不同的单词都有索引。如果单词“apple”对应于 0 索引，那么每次在文档中出现“apple”这个单词，就在这个索引处增加计数值。这最终变成了一个单词计数的练习：定义一组不同的单词，然后计算它们出现的次数。当有不同的单词计数时，可以根据单词索引映射到向量中的方式构建特征向量。在这个向量形式中，保持了单词出现的次数，但丢失了单词的排序信息。这种表示也可以被认为等同于直方图表示。

表8-5：向量空间模型的直观展示

	T_1	T_2	...	T_t
D_1	W_{11}	W_{21}	...	W_{t1}
D_2	W_{12}	W_{22}	...	W_{t2}
...	...	...		...
D_n	W_{1n}	W_{2n}	...	W_{tn}

通常将词袋模型称为**词频**向量，我们在 TF-IDF 这样更精细的向量化方案中使用词频。TF-IDF 将文档中单词的频率乘以它在整个语料库中出现的稀有度。在词袋模型的其他变体中，只使用 1 或 0 作为向量的条目值，用来表明单词是否出现。词袋模型需要数据集的多次传递，成本可能很高，尤其是对于较大的数据集。

词袋法的一个缺点是不适用于短语和多词表达。如果不经过特定的预处理，词袋无法处理单词可能的拼写错误或我们想拼出的单词变形。N-gram 和其他预处理技术在一定程度上解决了这两个问题[4]。

注 3：Zellig Harris 于 1954 年发表的关于“分布式结构”的文章是词袋模型的最早的出处之一（Salton 和 McGill的文章“Orderless document representation: frequencies of words from a dictionary”发表于1983年）。

注 4：词袋模型的另一个向量化变体是 hashing trick，它使用哈希函数将每个词映射到固定大小的向量中的索引。

8.5.2 TF-IDF

TF-IDF 是解决词袋模型一些固有问题的好方法。单词在所有文档或同一文档出现的频率不会相同。使用 TF-IDF 来测量单词出现频率与该词作为标记的独特性之间的相对权重。

TF-IDF 给出一个统计数字，来表示一个单词对于一个文档以及一个文档集合有多么重要。在 IR 和文本挖掘中，TF-IDF 经常用作权重因子。TF-IDF 值随文档中单词的出现次数成比例地增加，但同时随着整个语料库中词频的增加而减少。这有助于 TF-IDF 解释那些在文档中可能显得重要，但实际上在整个文档集合中却很普通的单词。

TF-IDF 已被证明是有用的，因为它允许我们使用单词在文档中出现的频率的信息，同时兼顾单词在语料库中的频率，以便应对某些单词更为常见的情况。TF-IDF 权重由词成分、词频（TF）和逆文档频率（IDF）组成，其中前者（TF）除以后者（IDF）。为了计算 TF-IDF 数值，从测量并计算“词频”部分开始。

词的权重

对于给定的用户查询，搜索引擎经常使用 TF-IDF 作为对文档相关性进行评分和排序的机制。TF-IDF 也可以用作文本摘要和分类的停止词过滤器。使用 TF-IDF，停止词将被赋予小的权重，而在整个语料库中很少出现的词被赋予更大的权重。

那些更重要的词通常 TF 更高、IDF 值更大，因此这两个因子的乘积更大。在 TF-IDF 的经典版本中，停止词被分配了一个小的权重，而很少出现的词被分配了一个更大的权重。这使得我们能够“看出”值相对较高的词，比如文档的主题。重要的词既有高的 TF 又有高的 IDF，其 TF-IDF 分数相对更高。

除了搜索引擎之外，TF-IDF 在文本摘要和分类领域也非常有用。这个向量化过程比基本的词袋模型更精确，但是计算起来更复杂，因为需要查看单词在文档和整个语料库中出现的频率，这涉及不止一个数据集和一些预处理的传递。

1. TF

TF 的定义是一个单词出现在一组文档中的次数的最简单形式。就像词袋表示一个单词在语料库中出现了一样，TF 表示单词出现的频率。对于更复杂的词频形式，需要根据文档长度将次数规范化。大多数文档的长度不会相同，根据这一点，将词频除以文档长度（文档中所有词的总数）的做法作为将次数规范化的一种方式，从而得到一个更精确的度量。

$$t = \text{词}$$

$$d = \text{文档}$$

$$tf_{t,d} = \frac{\text{count}(t)}{\text{count(alltermsind)}}$$

2. IDF

计算 TF-IDF 的下一步是通过计算单词的 IDF 来衡量单词提供了多少信息。我们想判断这个词有多么常见或少见，于是使用这部分的公式来计算，将得到的信息包含在向量中。这

么做的思路是那些出现在少数几个文档中的词比在语料库的许多文档中都出现的词应该在向量中分配一个更高的值[5]。

为了确定“文档频率”，首先通过将语料库中的文档总数除以包含该词的文档数来计算包含该词的文档分数。从这个值可以判断出，从考虑的整个文档组的角度来看，这个单词有多么少见。最后取这个导出分数的对数，得到的就是 IDF。

取对数之前的 IDF 值仍然不理想，因为它稀释了 TF 对最终词权重的影响。为了减少这个影响，通常用 IDF 值的对数来代替（表 8-6 列出了变量定义）。

$$\mathrm{idf}_t = \log\left(\frac{N}{df_t}\right)$$

表8-6：IDF组成部分

IDF变量	说明
df_t	文档频率；语料库中包含词 t 的文档数量
N	输入语料库中的文档总数

3. TF-IDF分数的完整计算

因此，一个词的 TF-IDF 权重的计算方式如下：

$$\mathrm{tfidf}_{t,d} = tf_{t,d} \times \mathrm{idf}_t$$

要想得到 TF-IDF 计算的高权重，单词需要相对于整个文档语料库具有相对高的 TF 和相对低的文档频率。往往把常见的词过滤掉，因为它们有低的 TF 和高文档频率。如果仔细查看权重的 IDF 部分，会发现它的值大于或等于 0。

这是由于输入经过对数函数计算的值总是大于或等于 1。当一个词出现在各个文档中时，对数内的比率接近 1。这将导致 IDF 值接近 0，进而导致 TF-IDF 值接近 0。实践证明，TF-IDF 比普通的词袋更高效，但是在预处理阶段，可以通过一种名为“N-gram”的技术提高效率。

DL4J 中包含了 TF-IDF 的实现[6]，作为其文本预处理工具的一部分。以下是使用 TF-IDF 的 DL4J 实现的简单示例。

```
File rootDir = new ClassPathResource("tripledir").getFile();
LabelAwareSentenceIterator iter = new LabelAwareFileSentenceIterator(rootDir);
TokenizerFactory tokenizerFactory = new DefaultTokenizerFactory();

TfidfVectorizer vectorizer = new TfidfVectorizer.Builder()
        .setMinWordFrequency(1)
        .setStopWords(new ArrayList<String>())
        .setTokenizerFactory(tokenizerFactory)
        .setIterator(iter)
        .build();

vectorizer.fit();
```

注 5：IDF 最初来自 Karen Sparck Jones 在 1972 年提出的“词的特异性”。

注 6：Apache Lucene 有一个 TF-IDF 的高质量的实现，它也在 JVM 中的许多地方被使用。



移除停止词

TF-IDF 中一个常见的预处理步骤是移除停止词。停止词占据原始文本的大部分，这会极大地影响 TF-IDF 权重的值。在 TF-IDF 中，为了在两个文档向量之间实现更精确的测量，可以移除停止词。停止词的例子包括：

- a
- an
- who
- the
- what

移除这些单词能极大地改进 TF-IDF 度量，因为两个文档之间的初始距离值由这些停止词的权重支配。

N-gram

N-gram 指来自给定文本或语音序列的 *N* 个连续项的序列。N-gram 向量化技术可以让向量化过程了解词组在语料库中一起出现的频率。一个基本的 TF-IDF 实现在向量化过程中不会拾取相邻单词包含的信息，并且可能遗漏如“Wall Street”或“Coca Cola”等标识中的依赖信息。

我们在那些有一个元素序列（如单词），并且我们想识别或预测下一个单词的任务中使用 N-gram。某些词在某些上下文中比其他词更可能出现，我们可以利用这些信息。N-gram 对一个词如何被它的一些相邻单词使用的上下文建模。N-gram 用于 NLP 和语音识别，在语音识别中，单词被建模为一系列 N-word。在语言识别的应用中，当我们试图发现构成文本的语言时，将字符序列（或字母，“字母表中的字母”）视为 N-gram。当解析文档或正文时，我们将其中的单词建模为每个 N-gram 由一个序列的 *N* 个单词构成。

对于 N-gram，需要注意的是，当 *N* 足够大时，运行算法的成本将变高。即使 $N = 5$，词汇量的增长也是相当惊人的，所以应谨慎使用 N-gram，并使用较小的 *N* 值。

核哈希

文本向量化中可以使用的另一种方法是核哈希。事实证明，向量化中的特征创建不是那么容易的，我们总是可以从简化事情当中受益。当处理大量数据时，在涉及大量硬件，同时试图获得良好的吞吐量的要求下，在 TF-IDF 中多次传递数据很困难。如果想在一次传递中将数据向量化，应使用核哈希，使之成为“即时”向量化器。使用核哈希的优点是不需要像 TF-IDF 那样多次传递，但是单词之间有可能发生冲突。可以在将文本向量化时，把文本提供给学习算法之前使用它。

8.5.3 Word2Vec与VSM的比较

如同第 5 章所讨论的，Word2Vec 通过学习周围单词的上下文，从数学上检测单词之间的相似性。Word2Vec 创建向量，这些向量是单词特征的分布式数字表示，例如单个词的上下文。具体说来，Word2Vec 生成一个词嵌入的列表（向量），其中单个向量表示每个单词。

使用 Word2Vec，可以通过理解单词在多个句子中的位置，并将该位置与意义联系起来，进而了解单词使用的语义。这使得 Word2Vec 向量能够基于单词神经网络的内部表示来推断输入数据中的潜在信息。



Word2Vec 向量算法

生成的词嵌入中词义和关系是空间编码的，并且具有向量算法等有用的特性。

相反，向量空间模型（如 TF-IDF）仅使用来自输入文档列表的同现统计来构建。这样会有一些有趣的特性，例如能够计算两个文档（或单词）之间的相似性，但是在表示单词或文档之间的语义或句法关系方面受到限制（只使用语料库中的同现信息）。

如前所述，Word2Vec 的向量表示使我们有新的方法来理解原始文本、进行推荐和图形建模。关于构造 Word2Vec（和段落向量）的代码示例，请参阅第 5 章。

8.6 使用图形

图形结构的向量化是工程数据科学流水线中的一个难题。虽然这里不会详尽讨论图形向量化，但会讨论 Node2Vec 的变体（也可以参考第 5 章）。

使用 Node2Vec，可以为一个图的每个顶点（并且扩展到每个边）生成向量。可以像（前面提到过）词向量一样将这些向量用于各种任务，例如节点分类和推断边是否应该存在（连接）。

Node2Vec 是一种被称为“DeepWalk”的图形方法的扩展。DeepWalk 通过在图形上随机漫步并执行截断的方式建立关于图结构的局部信息。这样 DeepWalk 就可以将对图的遍历访问当作处理句子一样，来学习图中的潜在表示。最后还希望你知道图形 CNN，但是 DL4J 框架目前还不支持它。

邻接矩阵

把图向量化为邻接矩阵的做法是机器学习中处理这种任务的一种常见方法[7]。在这种方法中，使用方阵表示有限图结构，其中矩阵的行和列都有图中所有节点的列表。邻接矩阵是一个 (0, 1) 矩阵，对角被填充为 0（这意味着在这种类型的结构中，节点的边不能连接到其自身）。

本书将更多地介绍能够直接在图形结构上操作的方法，而不是构建邻接矩阵的自定义变体。

注 7：类似的用于图形向量化的“邻接列表”数据结构还有很多。

第9章

在Spark上使用深度学习和DL4J

路途上度过的十年，让我一夜之间，
送走了年轻时光。
再告诉我一次，我就会明白，
你确定汉克就是这样做的吗?
老汉克真的就是这样做的吗?

——美国乡村歌手 Waylon Jennings，歌曲《你确定汉克就是这样做的吗？》

9.1 在Spark和Hadoop上使用DL4J的介绍

过去十年出现的两个关键数据中心技术是 Apache Hadoop 和 Apache Spark。尤其是 Hadoop，成为了数据仓库增长和演化的中心。Spark 继承了 MapReduce 的衣钵，成为在 Hadoop 上运行并行迭代算法的主流运行框架。

DL4J 支持在 Spark 上进行网络训练的扩展。使用 Spark 执行 DL4J 可以显著减少网络训练所需的时间。这种用法也使我们多了一种选择，即能够减少随着输入大小的增长而增加的训练时间。



上云

亚马逊云服务、谷歌云和微软 Azure 等平台让人们只需花几美元，就能按需建立 Spark 集群。DL4J 能够在大多数公共云基础设施上运行 [1]，给了实践者运行深度学习工作流时在方式和平台选择上的灵活性。

注 1：目前 DL4J 还不支持 DataBrick 的云 Spark 服务，但未来可能支持。

Spark 是一个常见的并行处理引擎，它可以单独运行，也可以在 Apache Mesos 集群上，或通过 Hadoop YARN（Yet Another Resource Negotiator：另一种资源协商器）框架在 Hadoop 集群上运行。使用 Hadoop 中包含的输入格式，它能够处理 Hadoop 分布式文件系统（HDFS）中的数据。Spark 使用一些技术，将常用的数据缓存到内存中的弹性分布式数据集（RDD，稍后介绍）。Spark 还允许程序员抽象并行处理，将更多的注意力放在手边的算法上。本书主要关注 Spark 的批处理能力，将它应用于并行迭代算法，比如 DL4J 中的 SGD。

以下是一个 Spark 作业的关键组件。

- 应用程序

  所编译的 Spark 作业的 jar 代表 Spark 应用程序。它可以是一个作业、链接在一起的多个作业，或者是一个交互式的 Spark 会话。

- Spark 驱动器

  Spark 驱动器运行 Spark 上下文并将应用程序转换成任务的有向图。这些任务在集群上被调度运行。每个 Spark 应用只有一个驱动器。

- Spark 应用主控器

  当通过 YARN 在 Hadoop 上运行 Spark 时，Spark 应用主控器与 YARN 在集群上协调资源。每个 Spark 应用程序都有一个应用主控器。

- Spark 执行器

  执行器在其运行的本地主机上执行单个 JVM 实例的多个任务。Spark 驱动器指示长期运行的执行器需要运行哪些任务。单个主机在其本地可以有多个 Spark 执行器。最终集群中可能有 100 多个甚至 1000 多个执行器，这些执行器分布在多台机器上，同时运行不同的 Spark 应用程序。

- Spark 任务

  Spark 任务表示执行器中处理部分分布式数据集的工作单元。

- RDD

  RDD 是可以并行处理的元素的容错集合。

RDD

RDD 是 Apache Spark 中的核心概念。这些数据集是可以并行处理的元素的容错集合。RDD 上的操作由高级语言代码编译，使得程序员可以更多地多关注算法和业务问题，而较少考虑执行方面的分布式系统管理的事情。

以下是在 Spark 程序中创建 RDD 的两种常见方法：

- 在 Spark 应用中并行化现有的集合；
- 引用外部存储系统中的数据集。

外部存储系统包括 HDFS、HBase、Cassandra 或与 Hadoop `InputFormat` 兼容的其他任何系统，实践中最常用的是 HDFS。

Apache Hadoop 包括一组并行处理工具（如 MapReduce）和用 Java 编写的分布式文件系统（如 HDFS）。它是基于谷歌的 MapReduce 和谷歌文件系统（Google File System，GFS）发布的设计。Hadoop 最初是为了并行化 Apache Lucene 搜索引擎的反向索引结构而构建的，是 Apache Nutch 项目的一部分。Hadoop 项目由 Doug Cutting 和 Mike Cafarella 开创，最终吹响了搜索引擎基础设施普及化的号角，随后引发了数据仓库构建方式的革命。2008 年 1 月，Hadoop 从 Nutch 项目中分离出来，成为了 Apache 的顶级项目。

雅虎公司内部采用该技术，并将急需的工程资源投入到项目中，进一步孵化了 Hadoop 项目。2008 年 4 月，Hadoop 声称创造了一项世界纪录：在仅仅 209 秒内（在一个有 910 个节点的 Hadoop 集群中）完成了 1TB 数据的排序。到 2009 年，许多公司，如 last.fm、Facebook、纽约时报，甚至田纳西河流域管理局，都把 Hadoop 看作使用普通的硬件对大量数据作并行处理的一种实用方法。今天，多个 Hadoop 分发供应商已经把 Hadoop 推广到财富 500 强企业及其他企业。Apache Hadoop 已经成为现代数据仓库的基础设施，并且是企业运行环境的事实上的标准。



DL4J，Spark 和 Hadoop

DL4J 从诞生之日起就被设计为 Hadoop 环境的一等公民。DL4J 可以在单独的 Spark 上、在 Mesos 的 Spark 上，或者通过 YARN 框架在 Hadoop 集群上的 Spark 上运行。使用 DL4J 进行实验的实践者也许只在 Spark 的集群上使用 DL4J，但是大型企业往往在 Mesos 上，或者 YARN 管理下所支持的 Hadoop 集群上操作 Spark。

从命令行操作Spark

通常从命令行运行 Spark 作业，并配置多个参数。下面解释从命令行运行 Spark 作业的基础知识，以及处理不同选项的方法。

1. spark-submit

`spark-submit` 的 bash 脚本是向集群提交作业的方式。以下是在 Hadoop 集群（CDH、HDP）上（通过 YARN）运行 Spark 作业的 `spark-submit` 命令及其所需的命令行选项。

```
spark-submit --class [class name] --master yarn [jar name]
    [job options]
```

下面是对前面的命令行中每个选项的说明。

- 类名（class name）
  这里指完全限定的类名。
- jar 名（jar name）
  jar 名指路径和 jar 文件名，也称“超级 jar”，它包含运行该作业所需的所有内容。
- 作业选项（job options）
  作业选项指 Spark 作业所使用的各种选项。

示例如下：

```
spark-submit --class io.skymind.spark.SparkJob --master yarn
    /tmp/Skymind-SNAPSHOT.jar /user/skymind/data/iris/iris.txt
```

在这个作业中，我们运行 jar Skymind-SNAPSHOT.jar 中的 io.skymind.spark.SparkJob 类，这个作业有一个参数详细描述了输入数据的位置。

如果想在运行时从命令行更改某些作业的配置属性，可以在命令行上放置额外的参数或在特殊的配置文件中指定它们，如下所示：

```
spark.master     spark://mysparkmaster.skymind.com:7077
spark.eventLog.enabled    true
spark.eventLog.dir        hdfs:///user/spark/eventlog
# Set spark executor memory
spark.executor.memory     2g
spark.logConf             true
```

像前面示例中的那样，将配置关键值放入文本文件中是实用的做法，这样可以使作业执行得更快，更易于管理。通常这个文本文件放在作业 jar 所在的本地目录中，并在从命令行运行作业时引用它。

2. 使用Hadoop安全和Kerberos

Kerberos 是一种常见的企业级认证系统。Kerberos 提供了免受诸如拦截认证等攻击的支持。它还可以通过不在网络中直接用明文发送凭据的做法，帮助系统阻止用户伪造请求的威胁。（如果你对安全相关内容不感兴趣，可转到 9.2 节。）

主流的 Hadoop 发行版，如 CDH 和 HDP 均支持 Kerberos 身份验证。



在 Hadoop 发行版上运行的验证

Skymind（对 DL4J 的商业支持）经过验证，确保新版本的 DL4J 能够在新版本的 CDH 和 HDP 上运行。

可以选择将 Kerberos 凭据存储在轻量级目录访问协议（LDAP）或活动目录服务（Active Directory，Windows 操作系统的组件）中。

要在 Kerberos 管理下的 YARN 集群上运行 Spark，需要做以下两件事：

(1) 手动将打包好的 Spark jar 上传到 HDFS；
(2) 在命令行初始化 Kerberos。

上传 Spark 打包文件。首先手动将打包好的 Spark jar 上传到 HDFS 目录：

```
/user/spark/share/lib
```

打包好的 Spark jar 通常位于本地文件系统的以下目录中：

```
/usr/lib/spark/assembly/lib
```

或者，在 CDH 上的目录是：

```
/opt/cloudera/parcels/CDH/lib/spark/assembly/lib
```

当在 HDP 上运行 Spark 作业时，库被上传到 HDFS，因此运行该作业的用户需要有对 HDFS 进行写入的权限（http://bit.ly/2tZVQzw）。



关于 Kerberos 和 Spark 的说明

在 Kerberos 管理的集群中，如果在没有手动将 jar 上传到 HDFS 的情况下运行 Spark 作业，那么作业将失败，上传 jar 的命令（作为作业的一部分运行）将静默失败，因为 Spark 对 Kerberos 的支持是有限的。

初始化 Kerberos。要初始化 Kerberos，首先输入以下内容：

```
kinit [user-name]
```

你将会被提示输入密码。在 Kerberos 初始化之后，可以用以下命令确认是否有 Kerberos 票据。

```
klist
```

其输出的显示内容如下所示，从输出中可以看出 Kerberos 票据是活动且有效的。

```
[skymind@sandbox ~]$ klist

Ticket cache: FILE:/tmp/krb5cc_1025

Default principal: skymind@HORTONWORKS.COM


Valid starting     Expires            Service principal

07/05/16 20:39:08  07/06/16 20:39:08  krbtgt/HORTONWORKS.COM@HORTONWORKS.COM

    renew until 07/05/16 20:39:08
```

9.2 配置和调优Spark运行

Spark 可以在不同类型的分布式平台上运行，也可以在本地单机上运行。本章重点介绍在基于 YARN 的 Hadoop 集群和基于 Mesos 的集群上运行的 Spark。Cloudera 的 CDH 或 Hortonworks 的 HDP 发行版的企业用户以及 Apache Mesos 集群管理系统的用户应该对这些系统很熟悉。

Spark 能够以不同的方式执行，这取决于执行是否是分布的，以及作业运行时 Spark Driver 进程在哪里执行。了解这些关于 Spark 运行的基本概念将有助于你从在本地运行的简单作业迁移到在集群上运行长期作业，可以断开与集群的连接，让作业夜间在集群上运行。

每个 Spark 应用程序都有一个可以在前台（客户端模式）或后台（集群模式）运行的驱动进程。假如关闭了前台客户端，作业将停止，因为我们停止了控制集群上作业的本地进程。但是可以退出后台客户端并让 Spark 作业继续运行，因为作业控制器在另一个主机上运行。这个活动的驱动进程被 Spark 用来管理作业流程和调度 Spark 任务。

在客户端模式下，Spark 驱动进程在它被执行的本地机器上运行。在集群模式下，Spark 驱动进程在集群本身的远程机器上运行。

9.2.1 在Mesos上运行Spark

你可能青睐在 Spark 支持的 Apache Mesos 集群管理器下以分布式模式运行 Spark，而不使用 Hadoop 发行版。Mesos 是用户从物理机器本身抽象出 CPU、内存、存储和其他计算资源的一种方法。它允许我们将机器集群视为单个逻辑资源池，从而能够更高效地运行支持容错的多租户系统。

当使用 Mesos 管理集群时，Mesos master 将代替 Spark master，成为集群的管理者。在这种模式下，当驱动程序创建 Spark 作业和要调度的任务时，Mesos 将确定哪个机器处理哪个任务。Mesos 集群上将存在许多短期任务，并且 Mesos 能够分配集群上那些使用其他框架的多租户的任务。

在 Mesos 上使用 Spark 有两种主要模式：

- 客户端模式
- 集群模式

在客户端模式下，Spark Mesos 框架直接在客户端机器上启动，并等待用户输入命令或运行程序。用户将直接在屏幕的控制台上看到驱动程序的输出。

要在客户端模式下使用 Spark 在 Mesos 上运行一个作业，需要执行以下步骤：

(1) 在 spark-env.sh 中设置某些与 Mesos 有关的环境变量；
(2) 将正确的 Mesos 集群 URL 传递到 SparkContext。

在集群模式下，Spark 驱动器在集群本身的主机上启动，用户可以从 Mesos Web UI 看到作业的结果。在集群模式下，需要通过 sbin/start-mesos-dispatch.sh 脚本以及 Mesos 主节点的正确 URL 地址，在集群中启动 `MesosClusterDispatcher`。可以像以前一样使用 spark-submit.sh 脚本，但是需要包括 `MesosClusterDispatcher` 主节点的 URL，示例如下所示：

```
./bin/spark-submit \
  --class io.skymind.spark.mesos.MyTestMesosJob \
  --master mesos://210.181.122.139:7077 \
  --deploy-mode cluster \
  --supervise \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /tmp/mySparkJob.jar \
  1000
```



要了解更多关于在 Mesos 上使用 Spark 的信息，请查看 Apache 网站介绍 Spark 的页面的“在 Mesos 运行”一节。该页面有一个完整的清单，列出了 Mesos 作业的关键配置选项。

还有一种场景，你可能想在同一台机器上将 Spark 和 Mesos 与现有的 Hadoop 集群一起作为单独的服务运行。在此场景中，Mesos 上的 Spark 作业可以通过 HDFS 中的完全限定 URL 地址来访问 HDFS 中的数据。

9.2.2 在YARN中执行Spark

YARN 是对通用目的的应用程序公开的 Hadoop API，允许程序作为一等公民和 MapReduce 一起运行。目前 YARN 上的 Spark 实现支持以下面两种模式执行 Spark 作业：

- yarn-client
- yarn-cluster

YARN 应用包含 `ApplicationMaster` 和 `NodeManager` 的概念。

> **YARN、`ApplicationMaster` 和 `NodeManager`**
>
> `ApplicationMaster` 跟踪 Hadoop 集群中特定作业的资源和执行情况。`NodeManager` 分配执行作业任务的 YARN 容器。本节的目的不是向你灌输更多的分布式系统术语，而是提供一些对 Hadoop 生产集群内所发生的事情的说明。
>
> 在执行 Spark 时，Spark 执行器在 YARN 容器内运行，Spark 作业由 `Spark ApplicationMaster` 协调。你只需要意识到这些过程，以更好地了解它们如何、在哪里、何时运行 Spark 作业。

每个 Spark 执行器作为 YARN 容器运行[2]。Spark 在同一个容器中承载多个任务，使任务启动快了几个数量级。

下面详细了解每种模式。

- ❑ yarn-client

  在 yarn-client 模式下，Spark 驱动器在提交作业的机器上执行。很多时候是从开发者的本地笔记本电脑通过网络连接到集群。

  Spark 驱动器与 Hadoop 集群中的 Spark `ApplicationMaster` 通信，并发出指令，这些指令将在 YARN 容器中的 Spark 执行器上作为任务执行。

- ❑ yarn-cluster

  Spark 驱动器在基于 YARN 的 Hadoop 集群（HDP[3]、CDH）中的 `ApplicationMaster` 上远程运行。YARN `ApplicationMaster` 从 YARN 获取输入，协调作业，也运行 Spark 驱动器进程。以这种方式运行的 Spark 作业允许用户关闭自己的客户端或终端会话，而仍然能完成作业。

Spark执行模式的比较

表 9-1 给出了在 YARN 的不同模式下运行 Spark 的快速总结。这张表最初是由 Sandy Ryza 为 Cloudera 的技术博客（http://bit.ly/2tZYx4r）编写的。

注 2：不要与 Linux 容器的概念混淆，这是一个不同的概念。

注 3：参考 http://bit.ly/2tQmuuh，然后从 NiFi 启动远程 Spark 作业。

表9-1：理解YARN支持的各种Spark运行模式

	YARN cluster	YARN client	Spark独立运行
驱动程序哪里运行？	ApplicationMaster	客户端	客户端
谁请求资源？	ApplicationMaster	ApplicationMaster	客户端
谁开始执行进程？	YARN NodeManager	YARN NodeManager	Spark worker
持久化服务	YARN ResourceManager 和 NodeManager	YARN ResourceManager 和 NodeManager	Spark Master 和 worker
是否支持 Spark Shell？	否	是	是



何时使用哪种 YARN 上的 Spark 运行模式

当在 YARN 上交互调试 Spark 作业时，我们可能使用 yarn-client 模式。对于长期运行或计划在生产环境运行的 Spark 作业，我们可能使用 yarn-cluster 模式。

为什么要在 YARN 或 Mesos 上运行 Spark

Hadoop 中的 YARN 框架允许 MapReduce 以外的应用程序利用 Hadoop 中的执行环境。在 YARN 上运行 Spark 使用户能够：

- 在不同的并发应用程序和框架之间动态共享和中心化配置同一个集群资源池；
- 利用 YARN 调度器，在应用程序之间实现更好的并发性；
- 动态分配和控制集群中单个 Spark 应用程序在任意给定时间使用执行器的数量；
- Kerberos 支持[4]。

能够在多个应用程序之间动态共享集群是企业生产系统的一个重要特性。这意味着可以在同一个集群上同时运行 Impala 查询、MapReduce 作业和 Spark 应用程序，并通过一些合理的方式控制每个应用程序消耗多少资源。使用 YARN 或 Mesos 调度器使我们可以在集群上一边运行即席查询，一边继续运行已调度的生产作业，并确保为生产作业分配了足够的资源，以履行所需的服务水平协议（SLA）。这是因为 YARN 以及 Mesos 都懂得以可预测和可控的管理资源的方式管理多个框架。

在同一集群上运行许多类型的工作负载的多租户环境中，YARN 和 Mesos 对 DevOps 最友好。

9.2.3 Spark调优简要介绍

概括地讲，可以从两个主要的调优方向来调整 Spark 作业：CPU 和内存。我们可以设置执行器的数量、每个执行器使用的内核数量，以及每个执行器处理时所需的内存量。

注 4：除了 Kerberos 支持外，Mesos 支持大部分操作。



Mesos 和 GPU

在 Mesos 下，也可以选择管理 GPU，而 YARN 目前还不支持 GPU。

1. 设置执行器的数量

可以用命令行参数或配置文件属性设置应用程序执行器的数量。设置命令行参数的方法如下：

```
--num-executor
```

设置配置文件属性的属性键如下所示：

```
spark.executor.instances
```

可以在 spark-defaults.conf 文件、另外的 Spark 配置文件或 API 中的 `SparkConf` 对象中设置它。

2. Spark执行器和CPU内核

Spark 应用程序中每个执行器都有相同数量的内核可用。可以用命令行参数或配置文件属性设置它。设置命令行参数的方法如下所示：

```
--executor-cores
```

设置配置文件属性的属性键如下所示：

```
spark.executor.cores
```

可以在 spark-defaults.conf 文件，另外的 Spark 配置文件或 API 中的 `SparkConf` 对象中设置它。



执行器内核和并发执行器任务

当在批处理模式下为 Spark 设置每个执行器使用的内核数量时，我们就有效地设置了要同时执行的任务数量。

3. Spark执行器和内存

类似于执行器控制内核的方式，可以通过以下方式控制每个执行器的堆大小。

```
--executor-memory
```

此外，也可以列出每个执行器分配的单位为兆字节（MB）或千兆字节（GB）的内存大小。也可以从属性配置文件中设置这个属性，如下所示：

```
spark.executor.memory
```

在 `spark.executor.memory` 中，有两种方法可以进一步控制 Spark 中的内存使用。

- `spark.shuffle.memoryFraction`
- `spark.storage.memoryFraction`

这两个设置控制执行器在转换数据和持久化数据之间划分内存。下面详细了解这两个设置。

- ❑ `spark.shuffle.memoryFraction`

  在 Spark 的混洗过程中，`spark.shuffle.memoryFraction` 控制聚合和组合所需的 Java 堆内存的数量。默认为 0.2，建议至少在早期保留这个设置值。

- ❑ `spark.storage.memoryFraction`

  `spark.storage.memoryFraction` 控制 Spark 管理缓存的 RDD 的总大小。默认值为 0.6，它控制 Java 堆内存中 Spark 所需的内存缓存的比例。这会确保执行器的内存使用不会超出 RDD 堆空间与这个值的积。

Spark 执行器也会有开销，使用以下设置来管理。

- ❑ `spark.yarn.executor.memoryOverhead`

  该设置控制着分配给每个执行器的非堆内存的量。此内存用于虚拟机开销、驻留字符串和其他本地开销。

  默认值为（executorMemory*0.10），最小值为 384MB。

4. Spark和YARN容器的资源分配

当在 YARN 上使用 Spark 时，必须考虑 YARN 可用的资源[5]，要考虑的主要 YARN 属性如下所示：

- `yarn.nodemanager.resource.memory-mb`
- `yarn.nodemanager.resource.cpu-vcores`

- ❑ `yarn.nodemanager.resource.memory-mb`

  `yarn.nodemanager.resource.memory-mb` 控制 Spark 集群中每个主机上容器使用的最大内存量。

- ❑ `yarn.nodemanager.resource.cpu-vcores`

  `yarn.nodemanager.resource.cpu-vcores` 控制 Spark 集群中每个主机上容器使用的最大内核数量。如果为 Spark 作业请求 10 个执行器内核，这将向 YARN 请求 10 个虚拟内核。

5. 理解执行器在YARN中的内部请求

参数 `spark.executor.memory` 控制执行器堆大小，但 JVM 也会使用非堆内存。为了计算完整的 YARN 内存请求，需要将 `spark.executor.memory` 和 `spark.yarn.executor.memoryOverhaul` 相加，以得到完整的 YARN 内存请求量。图 9-1 直观地显示了这一点。

注 5：还应注意，YARN 在分配上优于 Spark，因为 Spark 进程在受限的 YARN 容器中运行。

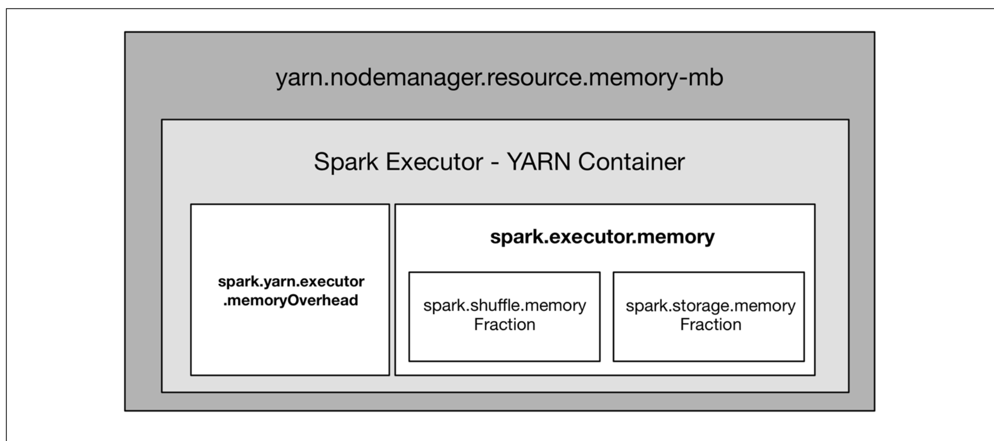


图 9-1：YARN 中 Spark 内存的层次结构

将 `spark.yarn.executor.memoryOverhead` 和 `spark.executor.memory` 的值相加，可得出 Spark 执行器完整的内存请求量。



注意需要请求多少内存

如果 `spark.yarn.executor.memoryOverhead` 和 `spark.executor.memory` 之和大于 YARN 中 `yarn.nodemanager.resource.memory-mb` 设置的值，那么作业将不会启动，因为它无法从 YARN 系统分配到所需的容器资源。

`spark.shuffle.memoryFraction` 的默认值为 0.2，`spark.shuffle.safetyFraction` 的默认值为 0.8。计算每个执行器任务可用内存的数量的公式如下所示：

```
( spark.executor.memory * spark.shuffle.memoryFraction
 * spark.shuffle.safetyFraction ) / spark.executor.cores
```

已经介绍了足够多调优 Spark 和 YARN 的内容，希望你可以就此开始了。需要注意，对 Spark 和 YARN 调优的设置还有很多，但本书不会涉及。



扩展阅读

阅读 Cloudera 的技术博客网站中的由 Sandy Ryza 写的博客（http://bit.ly/2tZYx4r），了解更多的 Spark 调优技巧。

6. 理解Spark、JVM和垃圾回收

Spark 下的 JVM 调优的主题庞大而复杂，基于本书的内容，我们总结了几个关键的主题，帮助你在需要的时候进一步了解。

处理垃圾回收速度慢或停顿的问题。实践者应该检查并确定 Spark 应用程序是否高效地使用了所分配的内存。如果 RDD 占用了大量内存，执行器的工作空间会变少，进而导致性能下降。

如果垃圾回收频繁发生，说明 Spark 需要更高效地使用内存。在已缓存的 RDD 不再使用以后，对其进行显示清理，来缓解这些问题。

为 JVM 和 Spark 选择垃圾收集器。 通常推荐对 Spark 执行器使用 G1 垃圾收集器。

9.2.4 对在Spark上运行的DL4J作业调优

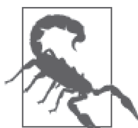
之前经讨论了 Spark 作业调优的基本机制，下面了解如何对在 Spark 上工作的 DL4J 作业调优。就像通常的 Spark 调优一样，我们可以控制 DL4J 作业中以下三个主要因素：

- worker（执行器）数量；
- 每个 worker 可使用的内存量；
- 每个 worker 可使用的内核数。

接下来介绍这些因素是如何影响深度学习模型的训练的。

1. 调整执行器的数量

当为 Spark 作业添加更多的执行器（worker）时，可以将数据分为更多的分区，并使每个 worker 处理更少的记录，这将减少总的训练时间。



减少回报与执行器

增加更多的执行器（超过某一临界点）会导致每单位训练时间可获得的准确度投资回报减少。尤其是在将 worker 的数据细分为小批量，使 worker 数量与数据分片相同之后，就不能继续细分了。增加 worker 将是无用的，因为 worker 将没有数据可处理。

2. 为执行器调整内存量

单个小批量的训练记录将作为单个矩阵或数据块发送到 ND4J，以向量化的方式被处理。这使得硬件使用更高效。控制小批量大小很重要，因为它不仅影响学习，而且影响执行器所需的内存量。

执行器需要处理的小批量的记录数量越多，需要分配给执行器的内存就越多。如果对执行器使用的内存有限制，必须注意不要将小批量设置得太多。



内存使用的经验法则

对于浮点数据（最常见）：每个值 4 字节，加上一些开销。下面是几个例子。

MNIST 数据（小）：28 × 28 ＝ 784 →每个样本约 3KB（加上标签：10 个值）。

具有 256 个输入、1000 个时间步的时间序列→每个样本特征 1MB（加上标签）。

理解并行性能

迭代算法的并行训练对收敛有一些小的的影响。DL4J 的分布式深度学习的用户可以选择小批量的大小，并且该软件将在 Spark 中的 worker 之间最优地分配工作负载。这类似于 Hadoop 的最优输入分片。系统将可用的处理单元的数量最大化，自动水平缩放。在 YARN 上，或在基于 Hadoop 的 Spark 等运行时上直接运行 DL4J，在基于 Hadoop 的 InputSplit（或 HDFS 数据块）系统数据集分片上运行 worker，这也将使每个 worker 的磁盘 I/O 得到良好的平衡。

在实践中，以参数平均为参数向量值的正则化器，通常建议首先增加 10% 轮数的训练用于额外的正则化。虽然训练论数略有增加，但得益于训练所用的 Spark 执行器的数量，执行每轮训练的时间会而减少。

一些设置（如罕见的平均周期）将需要更多轮的训练。对每个小批量取平均的做法（表现不佳，故不推荐）应该非常类似于本地训练（每次迭代的小批量大小等于所有执行器的样本总数）。

9.3 为Spark和DL4J建立Maven项目对象模型

建立 Maven 项目对象模型（POM）文件是构建 DL4J 项目、Hadoop 作业或 Spark 作业的关键部分。Apache Maven 允许你将所有需要的文件和依赖打包到单个 jar 模块中。本节将介绍在构建 DL4J POM 文件时的一些最佳实践。L4J Spark 应用的主要依赖如下所示：

- DL4J
- ND4J
- DataVec
- DL4J-Spark

下面介绍如何在 Maven 的 pom.xml 文件中进行这些操作。根据正在运行的 Spark 版本，需要以不同的方式配置 pom.xml 文件。为了支持与 Hadoop 的基本交互，添加以下 Maven 依赖：

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>
```

`hadoop.version` 变量要在 pom.xml 文件的属性区域中根据 Hadoop 发行版来设置。要查看更多关于 Maven 变量的值，可以查看 Hadoop 供应商网站（CDH：http://bit.ly/2uvDPGu；HDP：http://bit.ly/2sOME0L）中关于 Maven 模块版本的说明[6]。

注 6：有关各版本能否与 Hadoop 一起良好工作的更多信息，请访问 Apache Hadoop 网站（https://hadoop.apache.org/docs/r2.7.2/hadoop-project-dist/hadoop-common/Compatibility.html）。

要在 Spark 上使用 DL4J，需要引入 `deeplearning4j-spark` 依赖。

```
<dependency>
    <groupId>org.deeplearning4j</groupId>
    <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
    <version>${dl4j.version}</version>
</dependency>
```



`scala.binary.version`

在前面的代码中，注意其中的 `_${scala.binary.version}` 是 Maven 属性。它必须是 `_2.10` 或 `_2.11`，并且应该与正在使用的 Spark 版本相匹配。

Maven 变量 `spark.version` 将取决于运行的 Hadoop 发行版或 Spark 发行版，`scala.binary.version` 变量取决于所使用的是哪个版本的 Spark。表 9-2 是对这些变量的汇总说明。

表9-2：Maven pom.xml的核心条目

Maven变量	说明
`hadoop.version`	描述所使用的 Hadoop 发行版或版本（例如 CDH、HDP 等）
`scala.binary.version`	取决于所使用的是什么版本的 Spark
`spark.version`	取决于 Hadoop 发行版或 Spark 的发行版



平台依赖

如果你在一个平台上构建，但在另一个平台上部署，请使用 `nd4j-native-platform` 依赖作为替代，设置该属性会引入所有平台的本地二进制文件。

例如在 Macbook Pro 的 Maven 上构建项目，并且打算在运行 RedHat 的 Spark 集群的节点上运行作业，则需要将该属性设置为目标 RedHat。

9.3.1 一个pom.xml文件依赖模板

本节介绍如何设置 pom.xml 文件的依赖部分。首先是为依赖开发模板，然后介绍在具体的 Hadoop 发行版中如何配置变量。示例 9-1 提供了此设置的代码。

示例 9-1　DL4J 项目的 pom.xml 文件

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-native-platform</artifactId>
            <version>${nd4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-api</artifactId>
            <version>${nd4j.version}</version>
        </dependency>
```

```
            <dependency>
                <groupId>org.scala-lang</groupId>
                <artifactId>scala-library</artifactId>
                <version>${scala.version}</version>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <dependencies>

        <!-- Spark and Scala Dependencies -->
        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-mllib_${scala.binary.version}</artifactId>
            <version>${spark.version}</version>
            <scope>${spark.scope}</scope>
        </dependency>

        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>${scala.version}</version>
            <scope>${spark.scope}</scope>
        </dependency>

        <dependency>
            <groupId>org.apache.spark</groupId>
            <artifactId>spark-core_${scala.binary.version}</artifactId>
            <version>${spark.version}</version>
            <scope>${spark.scope}</scope>
        </dependency>

        <!-- Deeplearning4j Dependencies -->
        <dependency>
            <groupId>org.deeplearning4j</groupId>
            <artifactId>deeplearning4j-core</artifactId>
            <version>${dl4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.deeplearning4j</groupId>
            <artifactId>dl4j-spark_${scala.binary.version}</artifactId>
            <version>${dl4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
            <version>${nd4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-native-platform</artifactId>
```

```
    <version>${nd4j.version}</version>
</dependency>

<!-- DataVec Dependencies -->
<dependency>
    <groupId>org.datavec</groupId>
    <artifactId>datavec-api</artifactId>
    <version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.datavec</groupId>
    <artifactId>datavec-spark_${scala.binary.version}</artifactId>
    <version>${datavec.version}</version>
</dependency>

<dependency>
    <groupId>org.apache.avro</groupId>
    <artifactId>avro</artifactId>
    <version>1.7.1</version>
    <type>jar</type>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<!-- hadoop-mapreduce-client-app -->

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-app</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>${spark.scope}</scope>
</dependency>

<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
</dependency>

<dependency>
    <groupId>joda-time</groupId>
```

```
            <artifactId>joda-time</artifactId>
            <version>2.7</version>
        </dependency>

        <dependency>
            <groupId>org.apache.mrunit</groupId>
            <artifactId>mrunit</artifactId>
            <version>1.1.0</version>
            <classifier>hadoop2</classifier>
        </dependency>

        <!-- JCommander for parsing args -->
        <dependency>
            <groupId>com.beust</groupId>
            <artifactId>jcommander</artifactId>
            <version>${jcommander.version}</version>
        </dependency>

    </dependencies>
```

控制 jar 的大小

作业 jar 过大的原因之一是引入了在执行平台上已经可用的依赖。通过排除这些 jar，有时可以将 jar 大小减少 50% 到 80%，从而使编译、移动和执行作业更加容易。可以使用 <scope> 标签来控制 jar 包含哪些依赖，如下所示：

```
<scope>provided</scope>
```

用此方式设置作用域的依赖的例子如下所示：

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-common</artifactId>
    <version>${hadoop.version}</version>
    <scope>provided</scope>
</dependency>
```



序列化

Spark 仍然默认使用 Java 序列化，这通常被视作一种不好的做法。在示例 9-1 中，通过引入上面的 Kryo 依赖来减轻这一影响，如下所示：

```
<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-kryo_${scala.binary.version}</artifactId>
    <version>${nd4j.version}</version>
</dependency>
```

本章稍后讨论 ND4J 和 Spark 的相关常见问题时，将详细讨论这个依赖。

下面介绍如何在特定的 Hadoop 发行版中进行设置。

9.3.2　为CDH 5.x设置POM文件

本节给出了在 CDH 5.x 上构建 Spark 作业的关键组件及其版本号。示例 9-2 是作业属性的一个例子。

示例 9-2　用于构建 CDH 5.x 作业的 pom.xml

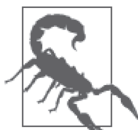
```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <slf4j.version>1.7.5</slf4j.version>
    <jackson.version>2.5.1</jackson.version>

    <hadoop.version>2.6.0-cdh5.5.2</hadoop.version>

    <!-- DL4J Versioning -->
    <nd4j.version>0.7.2</nd4j.version>
    <dl4j.version>0.7.2</dl4j.version>
    <datavec.version>0.7.2</datavec.version>

    <scala.binary.version>2.10</scala.binary.version>
    <scala.version>2.10.4</scala.version>
    <spark.version>1.3.1</spark.version>

</properties>
```



对于任何基于 DL4J 的项目，`nd4j.version`、`dl4j.version` 和 `datave.version` 的版本号都需要在 Maven 的 pom.xml 文件中相互匹配。

下面介绍 Horton 的 Hadoop 发行版中的 POM 文件。

9.3.3　为HDP 2.4创建POM文件

本节展示如何为 HDP 2.4 上的 Spark 作业配置 pom.xml 文件（假设你负责创建 pom.xml 样板），这些是作业的关键依赖和组件版本的设置。

示例 9-3 展示了用于 HDP Spark 作业的 Maven 的 pom.xml 模板的属性部分。这些变量将与下面的依赖项相匹配。

示例 9-3　用于构建 HDP 2.4 作业的 pom.xml

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <slf4j.version>1.7.5</slf4j.version>
    <jackson.version>2.4.4</jackson.version>
    <jcommander.version>1.27</jcommander.version>


    <!-- HDP 2.4 Version modify for older/newer HDP releases -->
    <hdp.version>2.4.0.0-169</hdp.version>
    <hadoop.version>2.7.1</hadoop.version>

    <spark.version>1.6.0</spark.version>
```

```
        <spark.scala.version>2.10</spark.scala.version>

        <!-- DL4J Versioning -->
        <nd4j.version>0.7.2</nd4j.version>
        <dl4j.version>0.7.2</dl4j.version>
        <datavec.version>0.7.2</datavec.version>

        <scala.binary.version>2.10</scala.binary.version>
        <scala.version>2.10.4</scala.version>

    </properties>
```

值得注意的是，构建系统使用示例 9-3 中的属性来构建 Maven 组件 jar。除了在 Maven 中使用之外，DL4J 项目代码本身没有使用这些变量。

9.4 Spark和Hadoop故障排除

本节列出了在 Hadoop 上使用 Spark 时会遇到的一些常见问题。

有时，如果执行器为每个容器请求的 RAM 比可用的多，那么 Spark 驱动器将会报告以下问题：

> 警告 TaskSchedulerImpl：初始作业没有接受任何资源，检查你的集群 UI，确保 worker 已注册并有足够的内存。

很多时候，可以通过降低作业请求的 RAM 或内核数量来缓解这种问题。

查看 Spark 调试信息的入口

表 9-3 列出了查看和调试 Spark 活动的一些关键端口。

表9-3：Spark调试信息关键端口

Web服务	端口号
YARN Job History Server UI	19888
YARN Resource Manager UI	8088
Spark Job History Web UI	18080

ND4J的常见问题

下面列出了在 Spark 中使用 ND4J 会遇到的一些常见问题，以及解决方案。

1. ND4J与Kryo序列化

Kryo 是一个常用于 Apache Spark 的序列化库。它的目标是通过减少序列化对象所花费的时间来提高表现。



SerDe

“SerDe”一词是数据的序列化和反序列化系统工程的简写。Spark 序列化涉及数据和函数。Spark 只关注设置序列化，且默认依赖 Java 序列化，这种做法方便但效率低。

Hadoop 引入了自己的 SerDe 机制（`Writable` 接口），并使用输入和输出格式，从文件格式解析 `Writable`，或者解析到文件格式中。Spark 需要这些输入 / 输出格式来操作 HDFS 中的数据。

然而，Kryo 难以处理 ND4J 中的非堆数据结构[7]。要在 Apache Spark 上配合 ND4J 使用 Kryo 序列化，需要对 Spark 进行一些额外的配置。如果 Kryo 没有被正确配置，那么由于不正确的序列化，有可能在某些 INDArray 字段上出现 `NullPointerException`。

要使用 Kryo，需要添加适当的 `nd4j-kryo` 依赖，并进行 Spark 配置以使用 ND4J Kryo 注册器，如下所示：

```
SparkConf conf = new SparkConf();
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.kryo.registrator", "org.nd4j.Nd4jRegistrator");
```



当使用 DL4J 的 `SparkDl4jMultiLayer` 或 `SparkComputation Graph` 类时，如果 Kryo 配置不正确，系统会记录一个警告。

2. jnind4j与java.library.path

以下是一个常见错误：

```
Exception in thread "main" java.lang.UnsatisfiedLinkError: no jnind4j in
    java.library.path
```

9.5 DL4J在Spark上的并行执行

作为加速网络训练的一种方式，DL4J 支持在 Spark 集群上训练神经网络。

类似于 DL4J 的 `MultiLayerNetwork` 和 `ComputationGraph` 类，DL4J 定义了用于在 Spark 上训练神经网络的两个类。

- `SparkDl4jMultiLayer`
  `SparkDl4jMultiLayer` 是对 `MultiLayerNetwork` 的封装。
- `SparkComputationGraph`
  `SparkComputationGraph` 是对 `ComputationGraph` 的封装。

因为这两个类是标准单机类的封装器，所以网络配置过程（例如创建 `MultiLayerConfiguration` 或 `ComputationGraphConfiguration`）在标准训练和分布式训练中相同。不过 Spark 上的分

注 7：这不是 ND4J 的问题，而是 Spark 的问题，但从实用的角度来看，这个问题值得注意。

布式训练与本地训练在两个方面不同：数据如何加载以及训练如何设置（需要一些额外的集群特定的配置）。

在 Spark 集群上（使用 Spark submit）训练网络的典型工作流如下所示。

(1) 创建网络训练类。通常涉及以下代码。

- 指定网络配置（`MultiLayerConfiguration` 或 `ComputationGraphConfiguration`），就像为单机训练所做的那样。
- 创建一个 `TrainingMaster` 实例,这指定了如何实际执行分布式训练（稍后将详细介绍）。
- 使用网络配置和 `TrainingMaster` 对象创建 `SparkDl4jMultiLayer` 或 `SparkComputationGraph` 实例。
- 加载训练数据。加载数据有许多不同的方法,有不同的取舍。更多细节见后面的章节。
- 在 `SparkDl4jMultiLayer` 或 `SparkComputationGraph` 实例中调用适合的 `fit` 方法。
- 保存或使用经过训练的网络（经过训练的 `MultiLayerNetwork` 或 `ComputationGraph` 实例）。

(2) 打包 jar 文件，用于 Spark submit。

- 如果使用的是 Maven，一种打包的方法是运行 `mvn package -DskipTests`。

(3) 选择适合集群的启动配置，然后调用 Spark submit。



单机上的 Spark 训练

对于单机训练，Spark local“可以”与 DL4J 一起使用，但不建议这样做（由于 Spark 的同步和序列化开销）。相反，请考虑以下做法。

- 对于单 CPU/GPU 系统，使用标准的 `MultiLayerNetwork` 或 `ComputationGraph` 进行训练。
- 对于多 CPU/GPU 系统，使用 `ParallelWrapper`。这在功能上等同于在本地模式下运行 Spark，不过它的开销更少（因此训练表现得更好）。

DL4J 的当前版本使用参数平均的过程来训练网络，未来版本可能还包括其他分布式网络训练方法。

使用参数平均来训练网络的过程在概念上相当简单。

(1) 以初始的网络配置和参数启动 master（Spark 驱动器）。
(2) 根据 `TrainingMaster` 的配置，将数据分割为若干子集。
(3) 对数据分片进行迭代。对于每个训练数据的分片：

- 将配置参数（以及可能存在的用于动量 /RMSProp/AdaGrad 的网络更新器状态）从 master 分发给每个 worker；
- 每个 worker 训练分配的数据；
- 平均参数（以及可能存在的更新状态），并将平均结果返回给 master。

(4) 训练完成后，master 会有一个训练网络的副本。

下面介绍这些概念在代码中是如何实现的。

最小的Spark训练示例

下面的代码展示了本章稍后将介绍的、构建完整 Spark 示例的基本概念。

```
JavaSparkContent sc = ...;
    JavaRDD<DataSet> trainingData = ...;
    MultiLayerConfiguration networkConfig = ...;

    //创建TrainingMaster实例
    int examplesPerDataSetObject = 1;
    TrainingMaster trainingMaster =
        new ParameterAveragingTrainingMaster
            .Builder(examplesPerDataSetObject)
            .(other configuration options)
            .build();

    //创建SparkDl4jMultiLayer实例
    SparkDl4jMultiLayer sparkNetwork =
        new SparkDl4jMultiLayer(sc, networkConfig, trainingMaster);

    //使用训练数据训练网络：
    sparkNetwork.fit(trainingData);
```

DL4J 中的 `TrainingMaster` 是一个抽象（接口），它允许使用 `SparkDl4jMultiLayer` 和 `SparkComputationGraph` 的多个不同的训练实现。目前 DL4J 有一个实现：`ParameterAveragingTrainingMaster`，它实现了前面代码中展示的参数平均处理。下面使用 builder 模式创建一个对象。

```
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(int dataSetObjectSize)
        ... (your configuration here)
        .build();
```

`ParameterAveragingTrainingMaster` 定义了一些控制训练执行的配置选项，下面详细介绍这些选项。

- `dataSetObjectSize`

  `dataSetObjectSize`（必选项）是在构造函数中指定的。这个值指定了每个 `DataSet` 对象中的样本数量。这里有两种情况要注意。

  - 如果使用预处理过的 `DataSet` 进行训练，值将是那些预处理过的 `DataSet` 的大小。
  - 如果直接从 `String` 类型数据（例如通过一系列步骤，从 CSV 数据转化为 `RDD<DataSet>`）进行训练，值通常为 1。

- `batchSizePerWorker`

  `batchSizePerWorker` 控制每个 worker 的小批量大小，这与在单机上训练时使用的小批量类似。换句话说，它是每个 worker 在每个参数更新时所使用的样本的数量。

- `averageFrequency`

  `averageFrequency` 控制参数根据小批量的大小 `batchSizePerWorker`，被平均和重新分配的频率。这里有三件事要记住。

- 低平均周期（例如 `averagingFrequency = 1`）可能是低效的（相对于计算而言，有过多的网络通信和初始化开销）。
- 高平均周期（例如 `averagingFrequency = 200`）可能导致表现不佳（每个 worker 实例中的参数可能显著不同）。
- 将平均周期设置为 5 到 10 个小批量范围内通常是一个安全的默认选择。

❑ `workerPrefetchNumBatches`

Spark worker 能够异步预取多个小批量（`DataSet` 对象），以避免等待加载数据。将该值设置为 0 禁用预取，而设置为 2 通常是明智的默认选择。在大部分情况下，更大的值不太可能有帮助（但会使用更多的内存）。

❑ `saveUpdater`

在 DL4J 中，将动量、RMSProp 和 AdaGrad 等训练方法称为"更新器"。这些更新器大多有内部历史或状态。如果 `saveUpdater` 被设置为 true，那么更新器状态（在每个 worker 中）将被平均，连同参数一起返回给 master，当前更新器状态也将从 master 分发到 worker。这增加了额外的时间和网络流量，但可以提高训练效果。如果 `saveUpdater` 被设置为 false，那么更新器状态（在每个 worker 中）会被丢弃，并且每个 worker 中的更新器会被重置 / 重新初始化。

❑ `repartition`

`repartition` 指定何时重新分区数据。`ParameterAveragingTrainingMaster` 执行 `mapPartitions` 操作，因此，分区的数量（以及每个分区中的值）对于正确使用集群非常重要。但重新分区不是一个无成本的操作，因为某些数据必须通过网络复制。以下选项可用。

- `Always`：（默认选项），即将数据重新分区以确保的分区数正确。
- `Never`：从不将数据重新分区，不管分区有多么不平衡。
- `NumPartitionsWorkersDiffers`：只有在分区数量和 worker 数量（内核总数）不同的情况下才重新分区。但是请注意，即使分区的数量等于内核总数，也不能保证每个分区中的 `DataSet` 对象数量都正确：一些分区可能比其他分区更大或更小。

❑ `repartitionStrategy`

`repartitionStrategy` 是重新分区的策略。以下是可选项。

- `SparkDefault`：Spark 所使用的标准的重新分区策略。基本上，初始 RDD 中每个对象分别被随机映射到 *N* 个 RDD 中的一个。因此，分区可能不是最优平衡的，对于较小的 RDD 问题可能比较严重，比如那些使用预处理的 `DataSet` 对象的 RDD 以及频繁的平均周期（仅根据随机采样变化）。
- `Balanced`：DL4J 的自定义重新分区策略。与 `SparkDefault` 选项相比，它试图确保每个分区（在对象数量方面）更加平衡。然而在实践中，这需要执行额外的计数操作。在某些情况下（最显著的是在小型网络中，或在每个小批量的计算量较小的网络中），它带来的好处可能不会超过执行更好的重新分区带来的额外开销。

9.6 Spark平台上的DL4J API最佳实践

以下是你应该做的、能够充分利用 Spark+Hadoop 集群的事情。

- 构建一个瘦身过的 jar。
- 使用调优良好的集群。
- 优化 ETL/ 向量化流水线。
- 确保 JVM 被调优过。

理想情况下，建立尽可能小的作业。扩展处理能力的核心理念是“将计算移至数据”，在集群中移动尽可能小的 jar。

应该在具有好的分布式文件系统且调优良好的集群中使用 Spark。

> **什么是“好的分布式文件系统”**
>
> 在分布式文件系统的领域中有几种选择，你应该考虑寻找一个知名的、支持可能用到的一组分布式组件的分布式文件系统。
>
> 数据科学家的工作内容不应该是搭建或者维护分布式系统的基础设施。由于基础设施的复杂性而混乱的机器学习项目是项目失败的“死神”。从这个角度，建议你选择具有以下特性的分布式文件系统：
>
> - 已知可靠；
> - 支持 Kerberos 集成；
> - 精心测试过；
> - 可扩展；
> - 可与数据科学堆栈中的大部分其他组件协同工作。
>
> 很多时候，特别是在 Spark 和 Hadoop 环境中，这个选择最终是 HDFS。

严肃的企业用户希望在一个维护良好的 Hadoop 集群中运行 Spark，最合适他们的是现代 Hadoop 发行版，我们推荐 CDH 5 或 HDP 2.4。

ETL 和向量化需要从主要训练循环中抽取出来。集群上的大型工作成本很高，所以它们需要尽可能的高效。理想情况下，应该保存和加载序列化 `DataSet` 对象，而不是一次又一次地转换 RDD。

最后，调优良好的 JVM 也是大有帮助的，让工作更轻松。确保 JVM 是调优后的状态，以免去长时间的垃圾收集暂停的困扰。

9.7 多层感知器的Spark示例

在这个例子中，我们将重新使用 MNIST 数据集来对数字进行分类，但是这次我们将使用多层感知器，并且将在 Spark 上训练网络。

本章的例子与第 5 章的例子相比，主要的改变是处理 Spark 上的并行训练的方式。本章的其他大部分例子都是类似的。

另一个需要考虑的方面是正在使用的 HDFS 中的数据，并且 ETL 和向量化流水线（通常）

需要以一种能够水平扩展的方式构建。可以使用 Spark 函数来完成这项工作（稍后介绍），或者可以使用 ETL 库，如 DataVec。本例使用一些自定义的内置代码来专门处理 MNIST 格式，因此不那么让人担心。需要注意的是，涉及 CSV 和基于文本数据的其他项目，可能需要大量的预处理。



小心单一处理的 ETL 代码

当处理更大的数据集时，需要注意构建 ETL 流水线的方式。Spark 允许我们将单一处理的 Java 代码与 Spark 函数混合，如果不够小心，我们将陷入窘境——代码无法根据输入数据集大小水平扩展。DataVec 在 GitHub 代码库中提供了 ETL 设计模式的示例（http://bit.ly/2sV9r6m），展示了如何在 Spark 中为 DL4J 工作流构建高效的并行 ETL 流水线。

示例 9-4 列出了 Saturn 示例的代码（已调整为可在 Spark 上执行）。

示例 9-4　用于 Spark 多层感知器网络的 Saturn 数据集

```
public class MnistMLPExample {
    private static final Logger log = LoggerFactory.getLogger(MnistMLPExample.class);

    @Parameter(names = "-useSparkLocal", description =
        "Use spark local (helper for testing/running without spark submit)",
            arity = 1)
    private boolean useSparkLocal = true;

    @Parameter(names = "-batchSizePerWorker", description =
        "Number of examples to fit each worker with")
    private int batchSizePerWorker = 16;

    @Parameter(names = "-numEpochs", description = "Number of epochs for training")
    private int numEpochs = 15;

    public static void main(String[] args) throws Exception {
        new MnistMLPExample().entryPoint(args);
    }

    protected void entryPoint(String[] args) throws Exception {
        //处理命令行参数
        JCommander jcmdr = new JCommander(this);
        try {
            jcmdr.parse(args);
        } catch (ParameterException e) {
            //用户提供了无效的输入 -> 打印使用方法信息
            jcmdr.usage();
            try { Thread.sleep(500); } catch (Exception e2) { }
            throw e;
        }

        SparkConf sparkConf = new SparkConf();
        if (useSparkLocal) {
            sparkConf.setMaster("local[*]");
        }
```

```
sparkConf.setAppName("DL4J Spark MLP Example");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

//将数据加载到内存中，然后并行化
//通常这不是一个好方法，但在这个例子中用起来很简单
DataSetIterator iterTrain =
    new MnistDataSetIterator(batchSizePerWorker, true, 12345);
DataSetIterator iterTest =
    new MnistDataSetIterator(batchSizePerWorker, true, 12345);
List<DataSet> trainDataList = new ArrayList<>();
List<DataSet> testDataList = new ArrayList<>();
while (iterTrain.hasNext()) {
    trainDataList.add(iterTrain.next());
}
while (iterTest.hasNext()) {
    testDataList.add(iterTest.next());
}

JavaRDD<DataSet> trainData = sc.parallelize(trainDataList);
JavaRDD<DataSet> testData = sc.parallelize(testDataList);


//----------------------------------
//创建网络配置，进行网络训练
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(12345)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .activation(Activation.LEAKYRELU)
    .weightInit(WeightInit.XAVIER)
    .learningRate(0.02)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .regularization(true).l2(1e-4)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
    .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
    .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
    .pretrain(false).backprop(true)
    .build();

//Spark训练的配置：参见http://deeplearning4j.org/spark，其中解释这些配置选项
TrainingMaster tm = new  ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker) //每个DataSet对象：包含（默认）32个样本
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2) //异步预读取：每个worker两个样本
    .batchSizePerWorker(batchSizePerWorker)
    .build();

//创建Spark网络
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

//执行训练：
for (int i = 0; i < numEpochs; i++) {
```

```
            sparkNet.fit(trainData);
            log.info("Completed Epoch {}", i);
        }

        //执行评估（分布式）
        Evaluation evaluation = sparkNet.evaluate(testData);
        log.info("***** Evaluation *****");
        log.info(evaluation.stats());

        //删除临时训练文件，现在已经完成了训练
        tm.deleteTempFiles(sc);

        log.info("***** Example Complete *****");
    }
}
```

下面剖析代码的关键部分，重点讲解示例的 Spark 版本中的核心差异。

要构建 Spark 作业项目，需要切换到项目的主目录，并运行下面的 Maven 命令：

```
mvn package
```

这将在 ./target 子目录中创建所需的 Spark 作业 jar。将这个 jar 复制到 Spark 集群的网关主机去运行作业，参考本章之前介绍的在命令行运行 Spark 的说明。

9.7.1 建立Spark MLP网络架构

示例 9-4 的网络架构类似于第 5 章中的某些网络。在下面的示例中，重点看看网络架构配置的代码。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .seed(12345)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .activation(Activation.LEAKYRELU)
    .weightInit(WeightInit.XAVIER)
    .learningRate(0.02)
    .updater(Updater.NESTEROVS).momentum(0.9)
    .regularization(true).l2(1e-4)
    .list()
    .layer(0, new DenseLayer.Builder().nIn(28 * 28).nOut(500).build())
    .layer(1, new DenseLayer.Builder().nIn(500).nOut(100).build())
    .layer(2, new OutputLayer.Builder(LossFunctions.LossFunction
        .NEGATIVELOGLIKELIHOOD)
        .activation(Activation.SOFTMAX).nIn(100).nOut(10).build())
    .pretrain(false).backprop(true)
    .build();
```

这个示例仍然使用 SGD 优化网络的参数。第 5 章的多层感知器示例只有一个隐藏层，而这个示例多了一个隐藏层。示例还切换到 ReLU 激活函数的 leaky 修正线性单元（ReLU）变体，这个变体在第 5 章的多层感知器示例中使用过。除此之外，网络和输出层的其余部分与另一个示例大体相同。这个示例很好地演示了如何通过简单地调整网络，来对完全不同的数据建模。诀窍是通过超参数搜索找出这些简单的调整。



本地和 Spark 上的相同网络架构

DL4J 的一个很好的特性是能够在本地机器上使用数据子集开发网络，然后将相同的结构移植到 Spark 上，并对整个数据集建模。

9.7.2 分布式训练与模型评估

下面是代码中要重点介绍的两个关键区别（与第 5 章的单一流程示例相比）。

- 在这个示例中引入了 `ParameterAveragingTrainingMaster` 方法。
- 不同的训练封装类：`SparkDl4jMultiLayer`。

与第 5 章的示例相比，这个示例中只有这两个区别是重大的修改。这些修改涉及的代码行数不多，因此可以看到从本地机器学习工作流迁移到 Spark 上的机器学习工作流并不是多么困难。下面的示例是示例 9-4 中引入的 `TrainingMaster` 的代码片段，如本章之面所解释的。在下一个代码片段中，使用它来控制在 Spark 中执行的参数平均。

```
//Spark训练的配置：参见http://deeplearning4j.org/spark，其中对这些配置选项进行了解释
TrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(batchSizePerWorker) //每个DataSet对象：包含（默认）32个样本
    .averagingFrequency(5)
    .workerPrefetchNumBatches(2) //异步预读取：每个worker 两个样本
    .batchSizePerWorker(batchSizePerWorker)
    .build();
```

Spark 上的分布式训练代码的另一个主要修改涉及一个名为 `SparkDl4jMultiLayer` 的 `MultiLayerNetwork` 封装器的使用，如下所示：

```
//创建Spark网络
SparkDl4jMultiLayer sparkNet = new SparkDl4jMultiLayer(sc, conf, tm);

//执行训练：
for (int i = 0; i < numEpochs; i++) {
    sparkNet.fit(trainData);
    log.info("Completed Epoch {}", i);
}
```

这个封装器与 `TrainingMaster` 协同工作，以执行模型所需的分布式训练，使我们得以从大量细节中抽身出来。`SparkDl4jMultiLayer` 类使我们能够以与在本地机器上执行模型训练几乎相同的方式控制执行器。

从前面的片段可以看出，`SparkDl4jMultiLayer` 类有三个参数：

(1) Spark context；
(2) DL4J 网络配置；
(3) `ParameterAveragingTrainingMaster` 对象。

在这个示例的本地机器版本中，可以看到用于对数据建模的 `MultiLayerNetwork` 类。`SparkDl4jMultiLayer` 封装器类的长处是它看起来几乎与 `MultiLayerNetwork` 类一模一样，可以在 `for` 循环中使用它，以相同的方式控制轮数。

最后，网络的 F1 分数被计算并打印到日志系统中。

```
//执行评估（分布式）
Evaluation evaluation = sparkNet.evaluate(testData);
log.info("***** Evaluation *****");
log.info(evaluation.stats());
```

这个示例非常好地展示了以干净、简单的方式使用 Spark 和 DL4J 训练模型的基本做法。除了增加几行额外的 Spark 代码和 `TrainingMaster`，以使深度学习模型能够在生产环境的、安全的 Spark 集群上训练之外，基本上不需要做别的事情。

9.7.3　构建和执行DL4J Spark作业

首先将当前目录切换为示例的根目录，接下来使用以下 Maven 命令构建作业 jar。

```
maven package
```

这将在 ./target/ 子目录下生成一个完整的作业 jar。

将作业 jar 复制到将要执行 Spark 的机器中之后，输入以下命令：

```
spark-submit --class org.deeplearning4j.examples.feedforard.MnistMLPExample
    --num-executors 3 --properties-file ./spark_extra.props
    ./dl4j-examples-1.0-SNAPSHOT.jar
```

这将输出大量训练信息到控制台，因为模型报告了训练的进展情况。注意，这里的命令行参数是那些由 `--properties-file` 参数提供的 Spark 运行时参数，这使得我们可以用常见的命令行参数并将它们保存到文件中，这样就不必每次都输入它们了。另一个需要注意的参数是 `--num-executors`，用于设置 Spark 中 worker 的数量，这里将其设置为 3。

9.8　使用Spark和LSTM生成莎士比亚作品

重温一下第 5 章的 LSTM 示例，看看如何修改代码，并在 Spark 上构建相同的模型（http://bit.ly/2sxFxFV）。下面的内容不会详细介绍数据加载的代码，而会重点介绍 Spark 下网络架构和训练过程的略微不同之处。示例 9-5 是核心的训练方法。

示例 9-5　使用 Java 开发的 LSTM 莎士比亚示例的核心训练方法

```
protected void entryPoint(String[] args) throws Exception {
    //处理命令行参数
    JCommander jcmdr = new JCommander(this);
    try {
        jcmdr.parse(args);
    } catch (ParameterException e) {
        //用户提供了无效的输入 -> 打印使用方法信息
        jcmdr.usage();
        try {
            Thread.sleep(500);
        } catch (Exception e2) {
        }
        throw e;
    }
```

```
Random rng = new Random(12345);
int lstmLayerSize = 200;                    //每个GravesLSTM层的单元数量
int tbpttLength = 50;                       //截断基于时间的反向传播的长度，即每50个
                                            //字符做一次参数更新
int nSamplesToGenerate = 4;                 //每次训练轮后生成样本的数量
int nCharactersToSample = 300;              //要生成的每个样本的长度
String generationInitialization = null; //可选的字符初始化；如果为null则使用
                                            //随机字符
//以上是用于"填充"LSTM的字符序列，以继续/结束。默认情况下，初始化字符都必须在
//CharacterIterator.getMinimalCharacterSet()中

//设置网络配置：
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
        .nOut(lstmLayerSize).activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX) //用于分类的MCXENT+softmax
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();

//-------------------------------------------------------------
//设置Spark特定的配置
/* 我们应该多久对参数取平均（根据小批量的数量）？频率太高会导致平均的计算
   变慢（同步+序列化成本），而频率太低又会导致学习困难（即网络可能不会收敛）
*/
int averagingFrequency = 3;

//设置Spark配置和上下文
SparkConf sparkConf = new SparkConf();
if (useSparkLocal) {
    sparkConf.setMaster("local[*]");
}
sparkConf.setAppName("LSTM Character Example");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

JavaRDD<DataSet> trainingData = getTrainingData(sc);

//设置TrainingMaster。TrainingMaster控制学习实际上是如何在Spark上执行的
//在这里使用标准参数平均
```

```
        //关于这些配置选项的详细信息，请参见：
        //https://deeplearning4j.org/spark#configuring
        int examplesPerDataSetObject = 1;
        ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
            .Builder(examplesPerDataSetObject)
            .workerPrefetchNumBatches(2) //异步预取两个批量
            .averagingFrequency(averagingFrequency)
            .batchSizePerWorker(batchSizePerWorker)
            .build();
        SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
        sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
            ScoreIterationListener(1)));

        //进行训练，然后生成并打印来自网络的样本
        for (int i = 0; i < numEpochs; i++) {
            //进行一轮训练。在每轮训练的最后，都会返回一个经过训练的网络副本。
            MultiLayerNetwork net = sparkNetwork.fit(trainingData);
            //从网络中采样一些字符（本地完成）
            log.info("Sampling characters from network given initialization \"" +
                (generationInitialization == null ? "" : generationInitialization) +
                    "\"");
            String[] samples = sampleCharactersFromNetwork(generationInitialization,
                net, rng, INT_TO_CHAR,
                nCharactersToSample, nSamplesToGenerate);
            for (int j = 0; j < samples.length; j++) {
                log.info("----- Sample " + j + " -----");
                log.info(samples[j]);
            }
        }

        //删除临时训练文件，现在已经完成了训练
        tm.deleteTempFiles(sc);

        log.info("\n\nExample complete");
    }
```

正如前面所做的，你需要自己研究其余的代码，为了简洁起见，这里不会列出数据加载的代码。

下面是代码中需要强调的两个关键差异。

- 在这个示例中引入了 `ParameterAveragingTrainingMaster`。
- 不同的训练封装器类：`SparkDl4jMultiLayer`。

与第 5 章的示例相比，这个例子中只有这两个区别是重大的修改，并且在前面的多层感知器 Spark 示例中有同样的修改。这些修改涉及的代码行数不多，因此可以看到从本地机器学习工作流迁移到 Spark 上的机器学习工作流并不是多么困难。为了加深理解，来具体看看在这个例子中，是如何定义 LSTM 网络的。

9.8.1 建立LSTM网络架构

下面强调的（来自前面）代码片段的有趣之处在于，它与第 5 章中单一流程的代码示例相同。

```
//配置网络：
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
        .iterations(1)
    .learningRate(0.1)
    .rmsDecay(0.95)
    .seed(12345)
    .regularization(true)
    .l2(0.001)
    .weightInit(WeightInit.XAVIER)
    .updater(Updater.RMSPROP)
    .list()
    .layer(0, new GravesLSTM.Builder().nIn(CHAR_TO_INT.size())
        .nOut(lstmLayerSize).activation(Activation.TANH).build())
    .layer(1, new GravesLSTM.Builder().nIn(lstmLayerSize).nOut(lstmLayerSize)
        .activation(Activation.TANH).build())
    .layer(2, new RnnOutputLayer.Builder(LossFunction.MCXENT)
        .activation(Activation.SOFTMAX) //用于分类的MCXENT+softmax
        .nIn(lstmLayerSize).nOut(nOut).build())
    .backpropType(BackpropType.TruncatedBPTT).tBPTTForwardLength(tbpttLength)
        .tBPTTBackwardLength(tbpttLength)
    .pretrain(false).backprop(true)
    .build();
```

这一点之所以很重要，是因为从操作的角度看，这是一个巨大的优势。可以使用在本地开发的网络配置，并使用 Spark 在 Hadoop 或 Mesos 集群上轻松地运行它们。

同样，这个例子中有两个隐藏层（`GravesLSTM`），它们都具有 tanh 激活函数，还使用了相同的自定义 LSTM 输出层（`RnnOutputLayer`），配合 MCXENT 损失函数和 softmax 激活函数。回顾一下第 6 章，任何时候想预测多个类别中的一个类别时，使用 softmax 激活函数，而这里要预测的是下一个字符（多个中的一个）。下面来看看这段代码与第 5 章的版本的细微差别。

9.8.2 训练、跟踪进度及理解结果

本章前面概括了将代码迁移到 Spark 执行时涉及的基本概念，当时重点介绍了以下（常见的）代码片段。

```
//创建TrainingMaster实例
int examplesPerDataSetObject = 1;
TrainingMaster trainingMaster = new ParameterAveragingTrainingMaster
        .Builder(examplesPerDataSetObject)
        .(other configuration options)
        .build();

//创建SparkDl4jMultiLayer实例
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, networkConfig,
    trainingMaster);
```

下一个代码片段将会展示，在一个新修改后的 LSTM 示例中，如何使用一个名为 `ParameterAverageTrainingMaster` 的特定 `TrainingMaster` 子类，以及如何在 `SparkDl4jMultiLayer` 对象实例中封装网络配置，以处理 Spark 集群操作的细微差别。

前一节讨论了使用 `TrainingMaster` 变体和 Spark 封装器的大多数可用选项，因此这里不再重复这些细节。

```
//设置TrainingMaster。TrainingMaster控制学习实际上是如何在Spark上执行的
//这里使用标准参数平均
//关于这些配置选项的详细信息，请参见：
//https://deeplearning4j.org/spark#configuring
int examplesPerDataSetObject = 1;
ParameterAveragingTrainingMaster tm = new ParameterAveragingTrainingMaster
    .Builder(examplesPerDataSetObject)
    .workerPrefetchNumBatches(2) //异步预读取两个批量
    .averagingFrequency(averagingFrequency)
    .batchSizePerWorker(batchSizePerWorker)
    .build();
SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, conf, tm);
sparkNetwork.setListeners(Collections.<IterationListener>singletonList(new
    ScoreIterationListener(1)));

//进行训练，然后生成并打印来自网络的样本
for (int i = 0; i < numEpochs; i++) {
    //进行一轮训练。在每轮训练的最后，都会返回一个经过训练的网络副本
    MultiLayerNetwork net = sparkNetwork.fit(trainingData);

    //从网络中采样一些字符（本地完成）
    log.info("Sampling characters from network given initialization \"" +
        (generationInitialization == null ? "" : generationInitialization) +
            "\"");
    String[] samples = sampleCharactersFromNetwork(generationInitialization,
        net, rng, INT_TO_CHAR,
        nCharactersToSample, nSamplesToGenerate);
    for (int j = 0; j < samples.length; j++) {
        log.info("----- Sample " + j + " -----");
        log.info(samples[j]);
    }
}
```

除了少数几行将示例移到 Spark 的代码之外，训练模型的循环都是类似的，都使用相同的方法（`.fit()`）来训练模型。与文件系统（如本地的或 HDFS）交互、与 Spark 模型通信和其他相关功能都在底层处理。

9.9 基于Spark上的CNN进行MNIST建模

对于这个示例，我们将基于第 5 章的 CNN 的例子来修改，使它在 Spark 上运行。示例 9-6 展示了用 Spark 构建 CNN 模型的代码。这个示例类似于本章前面的 MNIST 建模示例，但是这次网络架构更改为更适合图像数据的网络：CNN 架构。

示例 9-6 使用了 CNN 的 Spark MNIST

```
public class MnistExample {
    private static final Logger log = LoggerFactory.getLogger(MnistExample.class);

    public static void main(String[] args) throws Exception {

        //创建Spark context，并加载数据到内存
```

```
SparkConf sparkConf = new SparkConf();
sparkConf.setMaster("local[*]");
sparkConf.setAppName("MNIST");
JavaSparkContext sc = new JavaSparkContext(sparkConf);

int examplesPerDataSetObject = 32;
DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
List<DataSet> trainData = new ArrayList<>();
List<DataSet> testData = new ArrayList<>();
while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
Collections.shuffle(trainData,new Random(12345));
while(mnistTest.hasNext()) testData.add(mnistTest.next());

//获取训练数据。注意，对于实际问题不推荐使用并行化。
JavaRDD<DataSet> train = sc.parallelize(trainData);
JavaRDD<DataSet> test = sc.parallelize(testData);


//设置网络配置（和每个标准的DL4J网络一样）
int nChannels = 1;
int outputNum = 10;
int iterations = 1;
int seed = 123;

log.info("Build model....");
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations) //如上训练迭代
.regularization(true).l2(0.0005)
.learningRate(.01)//.biasLearningRate(0.02)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的过滤器的数量
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
        //注意nIn无须应用到后面的层
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
```

```
        .layer(4, new DenseLayer.Builder().activation(Activation.RELU)
                .nOut(500).build())
        .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
                .NEGATIVELOGLIKELIHOOD)
                .nOut(outputNum)
                .activation(Activation.SOFTMAX)
                .build())
        .setInputType(InputType.convolutionalFlat(28,28,1)) //查看下面的注释
        .backprop(true).pretrain(false).build();

        MultiLayerNetwork net = new MultiLayerNetwork(conf);
        net.init();


        //根据配置创建Spark多层网络
        ParameterAveragingTrainingMaster tm =
            new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
                .workerPrefetchNumBatches(0)
                .saveUpdater(true)
                .averagingFrequency(5)                          //每个worker执行5个
                                                                //小批量的训练，
                                                                //之后计算平均，
                                                                //并重新分布参数
                .batchSizePerWorker(examplesPerDataSetObject)   //每个worker在每次训
                                                                //练时使用的样本数量
                .build();
        SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

         //训练网络
         log.info("--- Starting network training ---");
         int nEpochs = 5;
         for( int i=0; i<nEpochs; i++ ){
             sparkNetwork.fit(train);
             System.out.println("----- Epoch " + i + " complete -----");

             //使用Spark评估网络：
             Evaluation evaluation = sparkNetwork.evaluate(test);
             System.out.println(evaluation.stats());
        }

        log.info("****************Example finished********************");
    }
}
```

下面详细讨论代码中特定的部分。

9.9.1 配置Spark作业和加载MNIST数据

使用 Java 版本的 Spark 执行框架需要创建一个 `SparkConf` 对象和一个 `JavaSparkContext` 对象实例。这是在本地模式或 Spark 集群中设置运行任务的地方 [8]。

注 8：若在本地运行，直接按原样运行示例即可。该示例默认在 Spark local 模式运行。注意：Spark local 模式应仅用于开发 / 测试。对于单机（例如多 GPU 系统）上的数据并行训练，使用 `ParallelWrapper`（在单机上它比直接使用 Spark 进行训练更快）。

当使用 Spark 时，数据通常需要存储在 RDD 结构中。下面的代码片段演示了如何使用第 5 章用过的迭代器获取原始 MNIST 数据，并将数据转换为 `JavaRDD` 实例。

```
DataSetIterator mnistTrain = new MnistDataSetIterator(32, true, 12345);
DataSetIterator mnistTest = new MnistDataSetIterator(32, false, 12345);
List<DataSet> trainData = new ArrayList<>();
List<DataSet> testData = new ArrayList<>();
while(mnistTrain.hasNext()) trainData.add(mnistTrain.next());
Collections.shuffle(trainData,new Random(12345));
while(mnistTest.hasNext()) testData.add(mnistTest.next());

//获取训练数据。注意，不推荐对实际问题使用并行化。
JavaRDD<DataSet> train = sc.parallelize(trainData);
JavaRDD<DataSet> test = sc.parallelize(testData);
```

将所有 MNIST 数据收集到内存中，然后使用 Spark context 对象创建 `JavaRDD`。

9.9.2　建立LeNet CNN架构与训练

与第 5 章中的 LeNet CNN 类似，前面的代码中出现了相同的架构配置。为了简单起见，再看一下这段代码。

```
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
.seed(seed)
.iterations(iterations) //如上训练迭代
.regularization(true).l2(0.0005)
.learningRate(.01)//.biasLearningRate(0.02)
.weightInit(WeightInit.XAVIER)
.optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
.updater(Updater.NESTEROVS).momentum(0.9)
.list()
.layer(0, new ConvolutionLayer.Builder(5, 5)
        //nIn和nOut指定深度。这里的nIn是nChannels，nOut是要应用的过滤器的数量
        .nIn(nChannels)
        .stride(1, 1)
        .nOut(20)
        .activation(Activation.IDENTITY)
        .build())
.layer(1, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(2, new ConvolutionLayer.Builder(5, 5)
        //注意nIn无须应用到后面的层
        .stride(1, 1)
        .nOut(50)
        .activation(Activation.IDENTITY)
        .build())
.layer(3, new SubsamplingLayer.Builder(SubsamplingLayer.PoolingType.MAX)
        .kernelSize(2,2)
        .stride(2,2)
        .build())
.layer(4, new DenseLayer.Builder().activation(Activation.RELU)
        .nOut(500).build())
```

```
    .layer(5, new OutputLayer.Builder(LossFunctions.LossFunction
            .NEGATIVELOGLIKELIHOOD)
            .nOut(outputNum)
            .activation(Activation.SOFTMAX)
            .build())
    .setInputType(InputType.convolutionalFlat(28,28,1)) //查看下面的注释
    .backprop(true).pretrain(false).build();
```

示例使用了与第 5 章的非 Spark 版本的示例相同的 `MultiLayerConfiguration`，并且层也相同，主要的区别在于如何控制 Spark 上训练循环的并行化，下面介绍它。

在 Spark 上训练 DL4J 模型与在单机上所做的事情类似，但也存在一些差异。在下面的代码中，来看一下示例代码的训练部分与第 5 章示例的不同之处。

```
//根据配置创建Spark多层网络
ParameterAveragingTrainingMaster tm =
    new ParameterAveragingTrainingMaster.Builder(examplesPerDataSetObject)
        .workerPrefetchNumBatches(0)
        .saveUpdater(true)
        .averagingFrequency(5)                          //每个worker执行5
                                                        //个小批量的训练，
                                                        //之后计算平均，并
                                                        //重新分布参数
        .batchSizePerWorker(examplesPerDataSetObject)   //每个worker在每次
                                                        //训练时使用的样本
                                                        //数量
        .build();

SparkDl4jMultiLayer sparkNetwork = new SparkDl4jMultiLayer(sc, net, tm);

//训练网络
log.info("--- Starting network training ---");
int nEpochs = 5;
for( int i=0; i<nEpochs; i++ ){
    sparkNetwork.fit(train);
    System.out.println("----- Epoch " + i + " complete -----");

    //使用Spark评估网络：
    Evaluation evaluation = sparkNetwork.evaluate(test);
    System.out.println(evaluation.stats());
}
```

这里的主要区别是（和前面两个例子一样）：

- 使用了 `ParameterAveragingTrainingMaster` 类；
- 使用了 `SparkDl4jMultiLayer` 类。

除此之外，代码并没有太大的区别。



Spark 代码与本地代码

DL4J 一个有用的特性是在本地机器上创建模型的方式与在 Spark 上创建模型的方式没有显著不同。正如本章所展示的，主要的差异并未涉及大量代码的修改。

附录A

人工智能是什么

Cooper：嘿，TARS，你的诚实参数是多少？

TARS：90%。

Cooper：90%？

TARS：绝对诚实并不总是最适合人际交往，也不是与有情感的人类最安全的交流方式。

Cooper：好的，90%很好。

——电影《星际穿越》中的一幕

人工智能是一门与哲学研究一样古老的学科。它随着时间不断演变，但是我们仍然在努力寻找它在社会中的位置，更不用说它对人类本身存在的意义。关于人工智能的开端，作家 Pamela McCorduck 的一段文字写得最好，她写道：人工智能始于“锻造众神的古老愿望”。

虽然 McCorduck 就这个话题写的是高尚的散文，今天的许多市场营销也都围绕着雄心勃勃的主题展开，但实际上在业务结果方面只产出了一些非常简单的功能。深度学习经常在对人工智能的讨论中出现，但很难就这个话题进行有针对性的交流。

之所以增加了本附录，是因为实践者们通常需要与客户、主管和管理层进行切实的交谈：深度学习能为他们做什么以及如何融入人工智能的世界。本附录包含对人工智能学科历史的介绍（供参考），以及我们与客户和业内同行的探讨。我们希望能够为作为实践者的你们提供工具，帮助你们重建对深度学习的论述，并与项目的利益相关者沟通，了解他们现实的期望并加以落实，这会更好地促进他们的深度学习成果向前发展。简而言之，围绕人工智能的论述已被夸大，最终必将被市场纠正。

本附录还以有趣和发人深省的方式，激发研究者或实践者的想象力，使我们心怀畅想，但

同时脚踏实地。我们将研究一些基本定义，简单回顾人工智能的历史，然后畅想一下人工智能未来走向何方。希望能够帮助人们避免掉入以前的人工智能兴趣陷阱，通过负责任地设置目标和期望，支持他们的深度学习项目更加成功。

A.1 历史回顾

本书的主题是深度学习，在媒体和市场营销中，它一直附属于术语“人工智能”。定义是易变的，很难与其他实践者或利益相关者讨论这个话题。营销部门则紧跟热点，什么热门就炒作什么。以下是不久之前的一些主题：

- 智能电网
- 云
- 大数据

当与这些领域或深度学习打交道时，作为实践者，需要从营销炒作中分辨出什么是真实的。这要求我们首先要了解该学科的历史，并打下坚实的基础。首先回顾一下什么是深度学习，然后深入讨论人工智能的定义。

A.1.1 定义深度学习

第 1 章和第 3 章给出了深度学习在工作上的定义，将其描述为具有以下特性的神经网络。

- 比以前的神经网络的神经元更多。
- 更复杂的连接层的方式。
- “寒武纪大爆发”般的用于训练的计算能力的增长。
- 自动特征学习。

这些网络和其他机器学习模型（回归、分类）执行相同的建模功能，但已显示出它们擅长下面这样的任务：

- 生成模型（例如生成艺术作品和文本）
- 语音识别技术
- 图像识别技术

深度学习的另一个关键驱动特征是它能够以无关领域的方式从数据中自动学习特征（与人工特征工程相反）。深度学习的这些能力正推动着许多新技术的应用，并激发了技术界以外许多人的想象力。然而，深度学习本身并没有更高级的功能，比如“自动理解询问数据集的最有趣的问题”，更不用说任何感知能力了。

A.1.2 定义人工智能

人工智能的历史充满了神话和传说，过于“热情”的营销部门总是试图利用最新的技术论述。为了定义人工智能，需要了解一些关于智能研究的历史、现代论点，以及该学科随时间演变的背景。

以这条线为出发点，我们来探索一下这个学科是如何开始的，以及如何在过去 60 年里作

为一个产业发展起来的。

1. 智能研究

智能研究于 1956 年正式在达特茅斯启动，但它至少已有 2000 年的历史。该领域基于对智能实体的理解，研究如下主题：

- 看见
- 学习
- 记忆
- 推理

我们认为，这些主题是智能所具有的我们可理解的能力的组成部分（这是一个相对的观点）。我们可以在历史中寻找到人们对智能的研究。下面列出的是一些随时间出现的智能研究的基石。

❑ 哲学（公元前 400 年）
  哲学家们开始把大脑设想为一种机械机器，它以某种形式对知识进行编码。

❑ 数学
  数学家发展了处理逻辑语句的核心思想，以及推理算法的基础。

❑ 心理学
  心理学这一研究领域建立在动物和人类大脑能够处理信息的观念之上。

❑ 计算机科学
  实践者提出用硬件、数据结构和算法作为对大脑逆向工程的基本组成部分。

今天看到的人工智能技术的研究和应用都是基于这些基本原理。人工智能的研究通常聚焦于模拟智能系统中的行为或思维。这些研究通常应用于机器学习应用、基本知识系统和游戏（例如国际象棋和围棋）等领域。

然而，智能研究的实现还存在局限性，还未出现具备高阶大脑功能（例如意识）的良好模型。科学还没有确定大脑意识的所在，这使得有些人质疑意识是否真的是大脑的功能。我们将这些质疑留给哲学家和计算机科学家们去辩论吧。



进一步学习

Stuart Russell 和 Peter Norvig 所著的《人工智能：一种现代的方法》是关于人工智能主题的最好的书之一（或去掉“之一”）。我们强烈推荐这本书，它能帮你更全面地了解人工智能的深度和历史。

2. 认知不一致与现代定义

在讨论人工智能等与社会基础的许多核心观念联系在一起的话题时，我们便会发现围绕基础真理建立起来的认知是不一致的。Beau Cronin 写道：

> 像物联网、Web 2.0 和大数据一样，人工智能被各种动机和背景的人们在许多不同的场景中讨论和辩论，这些人包括学者、商人、记者和技术专家。和其他定义模糊的技术一样，人工智能的含义也很难确定，每个人都能看到他想看到的东西。

我们的观点以及定义智能的部分问题是，我们天马行空地定义起了意识，然后扩展到更具哲学意味的问题（例如“什么是意识”）和宗教问题（例如“什么是灵魂”）。在这一点上，我们接触到了一些复杂领域，任何关于灵魂定义的讨论都是混乱且复杂的。时至今日，最好的定义了智能的地图都被“龙出没”的地区所铺满。假如我们不了解自然智能，就更难去定义人工智能了。

Jason Baldridge 博士写了一篇关于人工智能和机器学习的文章（http://bit.ly/2tUwIt5），并谈到了围绕着这个话题，冲突是如何产生的。

> 不管人工智能的技术定义有多么微妙，我敢肯定，当公众听到“人工智能”时，他们会想到有意识的非生物实体，它们与人类交互就像我们彼此交互一样。
>
> 他们没有想到一个专家系统可以分析复杂的领域特定问题，并提供有趣的行为过程，也没有想到机器学习算法可以在大量的数据中发现令人感兴趣的模式。
>
> 尽管如此，通常公众似乎都能在心理上很容易弥合这两种在人工智能相关工作领域非常不同的科技成就水平之间的差距。

Baldridge 博士接着定义了深度学习与生物大脑的全人工模型的区别。

> 所有这些进步，无论好坏，都离感官机器很远。深度学习的灵感来自人类神经元的功能，但是就我所知，人工神经网络还没有类似于生物智能的架构。

因此，定义这些很艰难，因为它们很复杂，有许多观点，触及许多主题。我们通过分解主题，并将这些细分为更简单的主题来寻找更好的定义。

Francois Chollet[1] 在推特上发表了以下重要观点：

> 人工智能是一个定义不清的东西，许多人认为它具有不受控制的、不切实际的能力，这会引起麻烦。

之后他再次发表推文：

> 部分问题是一些公司和新闻记者在炒作，模糊了科幻和现实之间的界限，因为这样有卖点。

接着 Chollet 说，我们应该“定义”正在谈论的内容：

> 当谈论“人工智能”时，定义你所谈论的内容，明确说明它能做什么，以及它不能做什么，避免脑补。

这是一个很好的建议，业界需要更好地锁定这些定义。

人工智能不是什么。声称机器学习是人工智能的人会对整个计算机科学行业造成伤害。机器学习就是分类和回归，它绝不是一个满足读者飘渺愿望的、全知的、具有自我意识的、有助于解决营销问题的系统。正如 Francois Chollet 之前提到的，（现在）最好避免脑补。

很多时候，人工智能都被营销为一个可以回答所有问题的应用。然而它不会，至少不会很快拥有这个能力。

注 1：谷歌研究员，Keras 创建者。——译者注

改变规则。心理学家习惯上不屑于将人脑比作计算机的说法。在 2016 年发表的一篇文章（http://bit.ly/2tABWqX）中，Robert Epstein 指出：

> 不管他们如何努力，脑科学家和认知心理学家永远不会在大脑中找到贝多芬第五交响曲的副本，或者单词、图片、语法规则或任何其他的环境刺激的副本。

不过，Epstein 博士还没有看到第 4 章中 CNN 过滤器的渲染。他在文章中的中心论点是：

> 你的大脑不处理信息、检索知识，或者存储记忆。简而言之，你的大脑不是电脑。

这并不是一种新的观点，大体上说，在过去的 60 年里，计算机科学以外的学科一直持有这样的观点。正如 Russell 和 Norvig 在关于人工智能的书中所说：

> 大体上知识分子们倾向于相信“机器永远不能做 X”。

他们继续展示了人工智能研究人员系统地回应的一个又一个 X 的例子。因此，对人工智能的研究和对学科的定义，长期以来深受业界“改变目标”做法的影响，偏离了“人工智能”的真正含义。

分解人工智能的定义。分解人们今天谈论人工智能的不同观点并列出它们，是有用的做法。Beau Cronin 写的一篇文章（http://oreil.ly/2sODKk2）中使用了以下四个主要的分解后对人工智能的定义。

- 作为对话者的人工智能：
  - HAL、Siri、Cortana、Watson；
  - 会话智能；
  - 有限推理。
- 作为机器人的人工智能：
  - 仿人型机器；
  - 机械形式的人工智能；
  - 终结者或 C3PO 这样的机器人；
  - 类似于对话者，但有人形的身体。
- 作为推理机的人工智能：
  - 早期的人工智能先驱们被更精致和高洁的任务所吸引——下棋、解决逻辑证明和计划复杂的任务；
  - 仍然在努力解决适合于孩子们的简单任务。
- 作为大数据学习者的人工智能：
  - 最近的定义；
  - 看到很多人在谈论构建“人工智能模型”。

下面我们用批判性的眼光来看一看这些分解后的定义。

对于定义分解的批判性评论。作为对话者的人工智能和作为大数据学习者的人工智能，都是将许多机器学习技术结合到商用产品中而得到的最新定义。作为对话者的人工智能可以基于语音识别来执行基本功能。它是语音到文本机器学习（或深度学习）和自然语言处理（NLP）技术的结合，以确定用户想完成什么。作为对话者的人工智能的推理能力有限，因为它通常依赖于一个单独的系统来发送语音到文本和以 NLP 处理的结果为输入。这个单独

的系统常常是经典的规则库的系统或“专家系统”的基础。

即使用户最初可能以与系统对话为乐，甚至被它的“智能”所骗到，但他们很快就会意识到交互的局限性。最终，作为对话者的人工智能是机器学习技术一个精心设计的组合，随着时间的推移，这些技术变得足够好，可以逐渐地集成到有用的消费品中。

作为机器人的人工智能是这个概念的一个有趣的体现，但是最终还是像对话者一样依赖机器学习子系统的网络。作为推理机的人工智能是人工智能的经典实现，但是近年来它停留在了人们想把它集成到工业产品的水平。它仍然是智能系统中的核心组件，将多个组件连接在一起以产生价值，就像对话者的例子一样。

“作为大数据学习者的人工智能”是一个引起争议的用语，在过去的几年（2010—2015）里已经流行起来。很多时候，市场营销部门会将产品中的基础机器学习技术对客户数据的使用重新命名为“人工智能”。更糟糕的是，其他时候，该产品仅仅执行基本的商业智能功能，但也被归为这一类。机器学习（或深度学习）单独的实践不应被视作一种人工智能，然而它是智能系统一个有用的子系统。



对深度学习的营销应表现得克制

在给机器学习模型（深层学习模型）贴上“人工智能”的标签时，应该表现得克制。过度炒作这个概念最初可能吸引到资金，但从长远来看会阻碍项目。

人工智能的第五个雄心勃勃的定义。另一种界定“什么是人工智能”的方法是问另一个问题。如果我们从“什么能终结‘什么是人工智能’的争论”的视角，如何看待这个问题？

如果出现一种超越人类的、有自我意识的智能，它比人类更好地理解我们的世界（和数据），我们可能称之为“真正的人工智能”，或者，外星人。

不幸的是，追逐人工智能的海市蜃楼是导致不切实际的期望的原因，不管取得了多少实质性的进步，这种期望总会压倒这个行业。

3. 人工智能寒冬

人工智能行业经历了多个兴趣和资金增加和减少的时期。这些兴趣减少的时期是该行业不切实际过度炒作的结果，随后是一连串可预测的低迷结果。这个低迷时期被称为“人工智能寒冬”，现象包括学术研究经费的削减，风险资本兴趣的降低，以及任何与“人工智能”一词沾边的事情都会被视为市场营销领域的耻辱。

这些周期导致的结果是，良好的技术进步（例如语音识别或光学字符识别）被重新命名，并被集成到其他产品中。

人工智能寒冬 I：（1974—1980）。第一个人工智能寒冬的开端是机器翻译没能引发热潮。人们对联结主义（神经网络）的兴趣在 20 世纪 70 年代衰退，而且对语音理解研究的过度承诺，也没有出现成果。

1973 年，DARPA 减少了人工智能领域的学术研究。英国的 Lighthill 报告严厉批评了该领域，导致研究经费被进一步削减。

人工智能寒冬 II：20 世纪 80 年代末。20 世纪 80 年代末和 90 年代初，诸如专家系统和

LISP 机器等技术被过度推广，这两项技术都未能达到预期。美国国家战略计算计划在这个周期结束时取消了新的支出。第五代计算机也未能实现其目标。

4. 人工智能寒冬的共同点

这些寒冬的共同点是行业出现了一系列所谓的有前景的、成功的过度炒作。当炒作得差不多之后，炒作进入谷底，学术和行业研究的资金涌入人工智能领域。一些基于可靠技术的实际项目至少达到了他们的部分目标，并解决了实际问题，然而市场上大多数的承诺都没能兑现，泡沫化的低谷期隐约可见。

冬天杀死弱者。

一些有趣的应用慢慢地从低谷中显现，被重新命名（如“语音识别”），并被作为特性集成到其他项目中，它们通常归于“潜在智能”改进分类中。这其中包括：

- 信息学
- 机器学习
- 知识库系统
- 业务规则管理
- 认知系统
- 智能系统
- 计算智能

名字的改变部分原因可能是他们认为该领域与人工智能根本不同，以及新的名称甩掉了“人工智能”虚假承诺的耻辱，有助于获得资金。

下面是 20 世纪 80 年代的一个人工智能会议中的有趣记录。

> 在会议上，Roger Schank 和 Marvin Minsky 这两位从 20 世纪 70 年代的“寒冬”幸存下来的人工智能研究领军人物警告业界：80 年代人们对人工智能的热情已经失控，失望肯定会接踵而至。三年后，十亿美元的人工智能产业开始崩溃。

A.2　如今驱动着人们对人工智能的兴趣的动力

如今有三大动力正在驱动着人们对人工智能的兴趣。

(1) 计算机视觉技术在 21 世纪 00 年代后期的巨大飞跃。
(2) 21 世纪 10 年代初的大数据浪潮。
(3) 顶尖科技公司在深度学习领域应用的进展。

2006 年，多伦多大学的 Geoff Hinton 和他的团队发表了一篇关于 DBN 的重要论文[2]，为业界提供了可能改善现有技术的创新火花。之后的十年里，在顶级期刊上出现了深度学习的滚滚浪潮。这些出版物提高了模型在许多领域中准确度的最高分数，不仅仅在计算机视觉领域，并且深度学习在短时间内占领了应用机器学习领域。

注 2：Hinton, Osindero, Teh. A fast learning algorithm for deep belief nets, 2006. https://www.cs.toronto.edu/~hinton/absps/fastnc.pdf.

大型网络公司如谷歌、Facebook 和亚马逊都在关注顶级期刊以获得最好的创意。这些公司看到了 Yann LeCun、Hinton 等开发的成果，并开始在自己的流水线中实施这些想法。这些新的应用（例如更好的人脸检测，或亚马逊的 Alexa）被技术媒体广泛认可。

21 世纪 00 年代中期，由西海岸的大型网络公司开发的许多存储和 ETL 技术开始开源，如 Hadoop 和 MongoDB。

这些网络公司（谷歌、雅虎等）已经增强了它们的存储和 ETL 系统，之后它们构建了新的机器学习和深度学习技术，以更好地利用这些新的更大的数据集。

传统的**财富 500 强**企业从 21 世纪 10 年代初开始引入在线大型分布式系统来保存不断增长的事务性数据集。这些企业计划用 5 到 10 年左右的时间来跟上西海岸的那些网络公司。这引起了人们在**财富 500 强**企业以及使传统企业更好地利用对大数据的投资的系统上应用深度学习的兴趣。

如果将前面的三个因素结合在一起，并结合 Watson（赢得 *Jeopardy* 竞答节目）、AlphaGo（赢得围棋比赛）和谷歌的自动驾驶汽车等非常成功的知名项目的实例，就会创造一个热情超出前方道路现实的环境。

对人工智能的报道和热情达到高潮。不幸的是，在一个又一个轮回中，我们也看到潮水最终退去。有些实际应用使用了复杂数据集进行深度学习。以下是部分这样的应用：

- 医疗保健（例如预测病人住院时间）
- 零售（例如分析购物体验）
- 电信 / 金融服务（例如分析欺诈模式的交易）

本书涵盖上述一些用例（以及更多其他的用例）。当作为实践者的你，在准备应用深度学习和人工智能时，建议去找诸如此类的真实用例，并且站在“坚实的基础”上。这里提到的“坚实的基础”是一个比喻，因为最终潮水会退去，希望我们的实践者同伴届时会有立足之地。

A.3　寒冬将至

在本书中，深度学习本身已经落地。它是一个用于在复杂数据类型上进行行业领先的神经网络建模的框架。不过深度学习本身不能满足前面提到的人工智能的第五个雄心勃勃的定义，因此我们在这方面没有什么可担心的。

2016 年被营销的系统是“人工智能 +X”，显然使用的是基本的机器学习。AlphaGo 是游戏领域的一个巨大的进步，但是正如在 IBM 深蓝和国际象棋的经历中所看到的，游戏领域的进步并不总是容易转化为商业用例[3]。

不幸的是，营销部门正在走前两个人工智能寒冬的老路。与之前的寒冬一样，促使该领域真正进步的“煤炭”会在即将到来的第三个人工智能寒冬中，使真正的热心者和核心研究人员保持温暖。

注 3：不过 *Jeopardy* 节目的问题似乎“已经解决了”。

附录B

RL4J与强化学习

作者：Ruben Fiszel

前言

本附录首先介绍**强化学习**，然后详细解释用于像素输入的深度 Q 网络（DQN），最后给出一个 RL4J 示例。下面介绍强化学习的核心概念。

强化学习是机器学习中一个令人兴奋的领域。它基本上是在给定环境中学习有效策略的过程。简单说来，它非常类似于**巴甫洛夫条件反射**：为给定的行为分配奖励，随着时间的推移，代理会学习复制这样的行为以获得更多的奖励。

马尔可夫决策过程

我们将这样的环境定义为**马尔可夫决策过程**（Markov Decision Process，MDP），它是五个元组的组合。

- 状态集合 S（例如在国际象棋中，状态是棋盘配置）。
- 可能的动作集合 A（在国际象棋中是每个可能的配置中每个可能的移动，例如 $e4$−$e5$）。
- 给定当前状态和动作，下一个状态的条件分布 $P(s'|s, a)$。（在国际象棋这样的确定性环境中，只有一个状态 s' 的概率为 1，而其他状态的概率为 0。然而在随机环境中，分布并不简单，如掷硬币。）
- 从状态 s 迁移到 s' 的奖励函数：$R(s, s')$（例如在国际象棋中，+1 表示最终导致胜利的移动，−1 表示最终导致失败的移动，否则为 0）。
- 折现系数：γ。比起对未来的奖励，这更是对当前的奖励（这个概念在金融领域很常见）。



使用动作集合 A_s

通常使用从给定状态开始的可用动作的集合 A_s 比使用完整集合 A 更方便。

A_s：A 中满足条件 $P(s'|s, a)$ 的元素 a。

马尔可夫属性（见图 B-1）是无记忆的，当到达一个状态时，过去的历史（以前访问过的状态）不影响下一个迁移和奖励，只有现在的状态才是重要的。

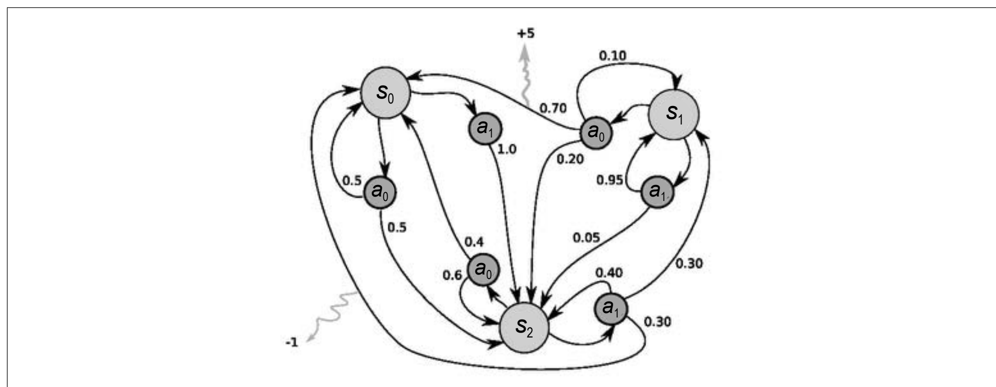


图 B-1：MDP 模式

术语

在继续学习之前，首先定义一些常见的单词和术语。

- 最终 / 终端状态

  没有可用动作的状态是最终 / 终端状态。

- 情节

  情节是从最初状态到最终状态的一个完整的剧本。

$$s_0, a_0, r_0, s_1, a_1, r_1, \cdots, s_n$$

- 累积奖励

  累积奖励是整个情节累积奖励的折现总和。

$$R = \sum_{t=0}^{n} \gamma^t r_{t+1}$$

- 策略

  策略是代理在每个状态选择行动的方式，用 π 表示。

- 最优策略

  最优策略是使累积奖励的期望最大化的理论策略。从期望值的定义和大数定律来看，在充分发生的情况下，这种策略的平均累积奖励最高。这种策略可能是可望而不可即的。

强化学习的目标是训练代理，使代理的学习策略尽可能接近最优策略。

B.1 不同的设置

“Qui peut le plus peut le moins.”（能难必能易。）

B.1.1 无模型

条件分布和奖励函数构成环境的模型。在西洋双陆棋的游戏中，模型是这样的：每个可能的移动是由已知的骰子分布决定的，并且我们能够在不去实际移动它们的情况下预测每次移动的奖励，因为我们可以计算棋盘的新值。TD-gammon 算法使用这个事实来学习 V 函数（参见 B.2 节）。

一些强化学习算法可以在不给出模型的情况下工作。然而，为了学习最优策略，它们在训练中还需要学习模型，这被称为**无模型强化学习**。无模型算法非常重要，因为现实世界中大多数复杂问题都属于这一类。此外，无模型只是一个额外的约束，它更加强大，因为它是基于模型的强化学习的超集。

B.1.2 观察设置

你也许只被允许观察状态的部分数据，而不能观察其全部数据。这与**隐马尔可夫链**背后的思想相同，这是部分和完全观察的设置之间的差异。例如我们的视野是对宇宙完整状态（宇宙中每个角落的位置和能量）很小的局部的观察。幸运的是，运用历史经验，我们可以将部分观察设置归纳为完全观察（状态变为先前状态的累积）。

然而，最常见的不是累积整个历史的数据，要么只（以窗口方式）堆叠最后 h 个观察状态，要么使用 RNN 来学习记忆什么以及忘记什么（这基本上就是 LSTM 的工作方式）。

为了与现有符号保持一致，我们稍微活用一下语言，将历史（甚至截断的历史）也称为“状态”，并且也标记为 S_t。

B.1.3 单人对抗游戏

单人游戏可以很自然地被翻译成 MDP（the moment during which the player is in control）。状态代表玩家控制的时刻，来自这些状态的观测值是状态之间累积的所有信息（例如控制帧之间所有的像素帧），动作是玩家可以使用的所有命令（在 Doom 游戏中是上、右、左、射击等）。

强化学习也可以自己玩对抗游戏：代理和自身对抗。在这种环境中，经常会存在纳什均衡，这样一来，你的对手就像一个完美的玩家，它总是符合你的兴趣。以对其有意义的国际象棋为例。给定一个棋盘布局，对阵一位象棋大师的一步好棋对于初学者来说依然是一步好棋。不管代理目前的水平如何，通过和自己比赛，代理依然能够知道它之前移动的质量（如果它赢了，就视为好动作，如果它输了，则视为坏动作）。

当然，如果从一开始就直接与一个非常好的代理较量，那么在神经网络的上下文中梯度信息的质量更高。但是，一个代理可以通过和自己，一个同样水平的代理比赛来学习提高自己的水平，这真是令人惊讶。这实际上是 AlphaGo（来自 DeepMind 公司，击败了世界冠

军的围棋代理）采用的训练方法。该策略以大师下棋的数据集为起点（最初训练），之后它使用强化学习以及与自己下棋的方式来进一步提高水平（用 Elo 算法评分量化），最后代理变得比它从原始数据集中学到的策略更好，毕竟它战胜了大师。为了计算最终的策略，AlphaGo 团队动用巨大的计算能力，并将策略梯度与蒙特卡洛搜索树相结合。

这个设置与从像素学习有点不同。首先，因为输入没有高维，所以流形更接近它的嵌入空间。然而在这种情况下，仍然使用卷积层来高效利用一些子网格棋盘布局的局部性。再则，因为 AlphaGo 不是无模型的（它是确定性的）。

B.2 Q学习

> 我不是概率论的朋友，从我们亲爱的朋友马克斯·玻恩给予它生命的那一刻起，我就讨厌它。因为虽然看起来它使每件事情都变得简单容易，但从原理上看，它把每件事情都熨平了，使真正的问题隐藏了起来。
>
> ——埃尔温·薛定谔

B.2.1 从策略到之后的神经网络

我们的目标是学习使以下表达式最大化的最优策略 π*。

$$E\left[R_0\right]=E\left[\sum_{t=0}^{n}\gamma^t r_{t+1}\right]$$

我们引入一个辅助函数：

$$V_{\pi}(s)=E\left\{r_t+\gamma r_{t+1}+\gamma^2 r_{t+2}+\cdots+\gamma^n r_n \mid s_t=s,\ \text{每个状态后的策略是}\pi\right\}$$

这是一个状态在策略 π 之后的期望累积报酬。假设有一个神谕，比如：

$$V_{\pi^*}(s)$$

它给出了最优策略的 V 函数。由此我们可以通过定义一个策略来检索最优策略，该策略从当前状态的所有可用动作中选择使$V_{\pi^*}(s)$的期望值最大化的动作。这是贪婪的做法，最优策略就是针对V_{π^*}的贪婪策略。

$$\pi^*(s)\text{chooses } a \ s.t\ a=\arg\max_a\left[E_{\pi}(r_t+\gamma V(s_{t+1})\mid s_t=s,a_t=a)\right]$$

细心的话你会发现这里可能有些问题。在无模型设置中，我们不能从 S_t 预测下一个状态 S_{t+1}，因为我们忽略了迁移模型。即使有了这个神谕，我们的模型仍然是不可计算的！

为了解决这个非常烦人的问题，我们将引入另一个辅助函数，Q 函数：

$$Q_{\pi^*}(s,a)=E_{\pi}\left[r_t+\gamma V_{\pi^*}(s_{t+1})\mid s_t,a_t=a\right]$$

在贪婪设置中存在以下关系：

$$V_{\pi}(s_t)=\max_a Q_{\pi}(s_t,a)$$

假设我们没有 V 神谕，但有 Q 神谕，现在可以重新定义 π*，如下所示：

$$\pi^*(s)\text{chooses } a\ s.t\ a = \max_a \left[Q_{\pi^*}(s,a)\right]$$

没有更多不可计算的期望，干净整洁。

然而我们只是把期望从神谕外部移到内部，并且现实世界中神谕并不存在。

这里的诀窍在于，我们已经将一个抽象的概念，即策略，简化为一个数值函数，并且该函数可能是符合预期的相对“光滑”（连续）的。对于我们来说幸运的是，我们可以使用神经网络来逼近这种复杂的函数。

神经网络是通用的函数逼近器，它们可以逼近任何连续可微的函数，然而它们可能陷入局部极值，并且当把神经网络放入方程中时，许多来自强化学习的收敛性证明不再有效。这是因为它们的学习不像其表格类型的同类学习那样具有确定性或有界性。但在大多数情况下，只要使用正确的超参数，它们都会极其强大。我们将深度学习与强化学习结合起来，称之为**深度强化学习**。

B.2.2 策略迭代

此时，你掌握的机器学习的知识和简单陈旧的常识可能告诉你，我们的方法仍然缺少一些东西。神经网络可以逼近已经有标签的函数，但神谕是不可召唤的，所以我们需要用另一种方式获得标签。

这就是蒙特卡洛方法的魅力所在，该方法依赖重复随机抽样来计算估计器（例如计算 π）。

如果我们从给定的状态随机地玩游戏，平均而言，更好的状态应该得到更好的奖励（这要感谢大数定律）。因此在不了解环境的情况下，你可以收集一些关于状态的期望值的信息。例如在扑克中，即使随机做出每个决定，平均而言，高手也会比庸手赢得更多。蒙特卡罗搜索树也是基于这个特性的（很惊讶，对吗？）。这是一种无监督学习的探索阶段，使我们能够提取有意义的标签。

更正式地，根据给定策略 π、状态 s 和动作 a，可以获得 $Q(s, a)$ 的近似值，根据定义对其进行采样。

$$Q_\pi(s,a) = E\left[r_t + \gamma r_{t+1} + \cdots + \gamma^n r_n \mid s_t = s, a_t = a\right]$$

简而言之，从状态 s 开始，可以根据策略 π 玩足够多的次数，来获得 $Q_\pi(s, a)$ 的标签。

来自一个信号集（见图 B-2）。

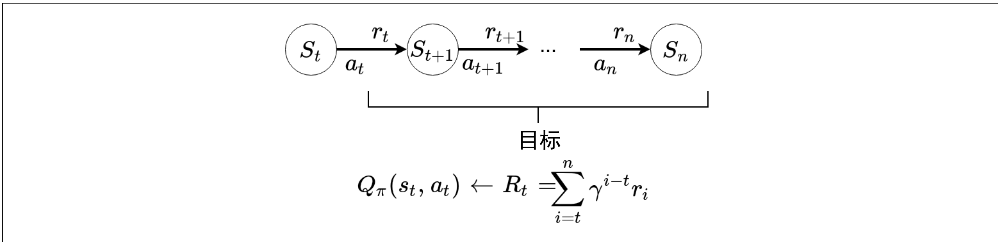


图 B-2：一个信号

实际的学习会使用批量标签，通过 SGD 完成。梯度是标准的均方误差（MSE），这使的每次迭代的 TD 误差最小化。

我们使用学习率为 α 的 MSE 损失函数（L2 损失），并且应用 SGD（批量大小为 1）。

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha[R_t - Q_\pi(s_t, a_t)]$$

(s_t, a_t) 是输入，$Q_\pi(s_t, a_t) + \alpha\left[R_t - Q_\pi(s_t, a_t)\right]$是标签（如目标）。



即使我们使用 MSE，公式中也没有平方，因为损失随后被应用到预期输出 $Q_\pi(s_t, a_t)$和标签$\alpha\left[R_t - Q_\pi(s_t, a_t)\right]$的差上。

重复多次，从 π 取样。

$$Q_\pi(s_t, a_t) \leftarrow E_\pi\left[R_t\right] = E_{s_t, a_t, \cdots, s_n \sim \pi}\left[\sum_{i=t}^{n} \gamma^{i-t} r_i\right]$$

图 B-3 可以收敛到正确的期望值。

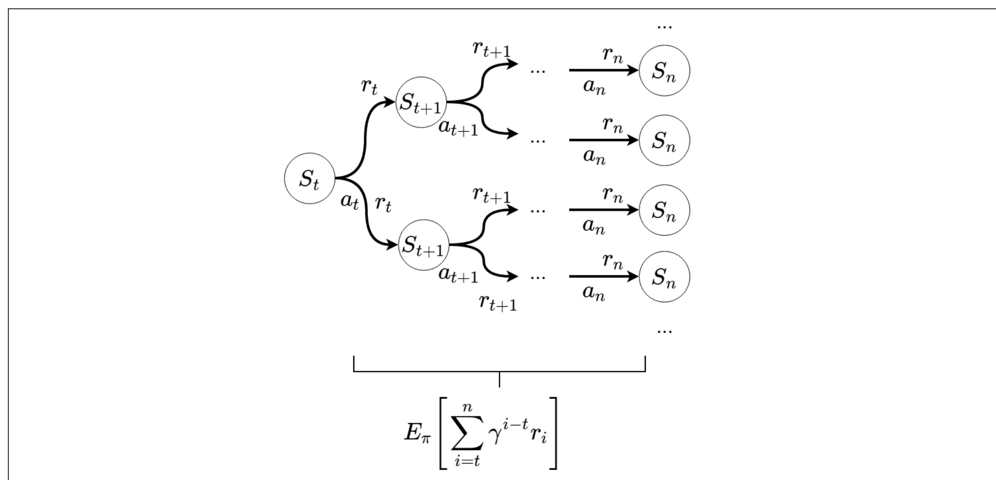


图 B-3：多个信号

因此，现在我们可以设计一个单纯的学习算法的原型（以 Scala 语言编写，但是不了解 Scala 也可以看懂代码），如示例 B-1 所示。

示例 B-1　用 Scala 编写的 RL4J 原型

```
//一个随机的未初始化的神经网络
val neuralNet: NeuralNet
//迭代，直到达到最大训练轮数
for (t <- (1 to MaxEpoch))
    epoch()


def epoch() = {
```

```
        //选择随机的状态和动作
        val state = randomState
        val action = randomAction(state)

        //迁移到新状态，初始化奖励
        var (new_state, accuReward) = transitition(state, action)

        //一直玩到结束状态，累积奖励
        accuReward += playRandomly(state)


        //对输入和标签执行SGD!
        fit((state, action), accuReward)
    }

    //MDP专用，返回新的状态以及奖励
    def transition(state: State, action: Action): (State, Double)

    //从所有状态空间中返回随机的样本状态
    def randomState: State

    //一直玩到结束状态
    def playRandomly(state): Double = {
        var s = state
        var accuReward = 0
        var k = 0
        while (!s.isTerminal) {
            val action = randomAction(s)
            val (state, reward) = transition(s, action)
            accuReward += Math.pow(gamma, k) * reward
            k += 1
            s = state
        }
        accuReward
    }

    //从该状态所有可用动作中随机选择一个动作
    def randomAction(state: State): Action =
        oneOf(state.available_action)

    //辅助函数，挑出一个动作
    def oneOf(seq: Seq[Action]): Action =
        seq.get(Random.nextInt(seq.size))

    //如何用DL4J大致地完成训练
    def fit(input: (State, Action), label: Double) =
        neuralNet.fit(toTensor(input), toTensor(label))


    //从ND4J返回INDArray
    def toTensor(array: Array[_]): Tensor =
        Nd4j.create(array)
```

这段代码存在多个问题：虽然能够工作，但效率非常低。我们正在玩一个其中单个标签有 n 个状态和 n 个动作的完整游戏，而且这个标签可能还没有意义（如果感兴趣的轨迹很难随机到达）。

B.2.3 探索与开采

随机探索，环境会收敛到最优策略，但这只有在几乎无限的时间之后才能得到保证：你将需要访问每个可能的迁移轨迹至少一次（轨迹是在一个情节中，所有访问的状态和选择的行动的有序列表）。考虑到状态和分支的数量，这是不可能的。分支问题是围棋比国际象棋更难的原因。在现实世界中，我们没有无限的时间（时间就是金钱）。

因此，我们应该利用过去的信息和我们对它的学习，把探索集中在最有前景的、可能的轨迹上。可以通过不同的方法达到这个目的，其中之一就是 ε-贪婪探索算法，它相当简单，策略是以 ε 的概率随机执行一个动作，或者以（$1-\varepsilon$）的概率执行当前策略认为最佳的动作。通常在充分探索之后，ε 随时间退火，使得相对于探索，开采被优先使用，这是在探索与开采之间的权衡。

随着每一条新信息的出现，实际的 Q 函数对于当前的策略变得更加精确，探索的重点是更好的路径。基于新的 Q 函数的策略变得更好（因为 Q 更精确），ε-贪婪探索找到更好的路径。Q 函数以这些更好的路径为核心，可以探索更好的部分，而且必须根据新的策略更新其回报。如图 B-4 所示，这是一个迭代周期，能够收敛到最优策略。因此把它称作**策略迭代**也就不足为奇了。遗憾的是，当用神经网络逼近 Q 时，收敛时间是无限长的，甚至不能保证收敛。然而，令人印象深刻的结果可以弥补形式收敛证明的不足。

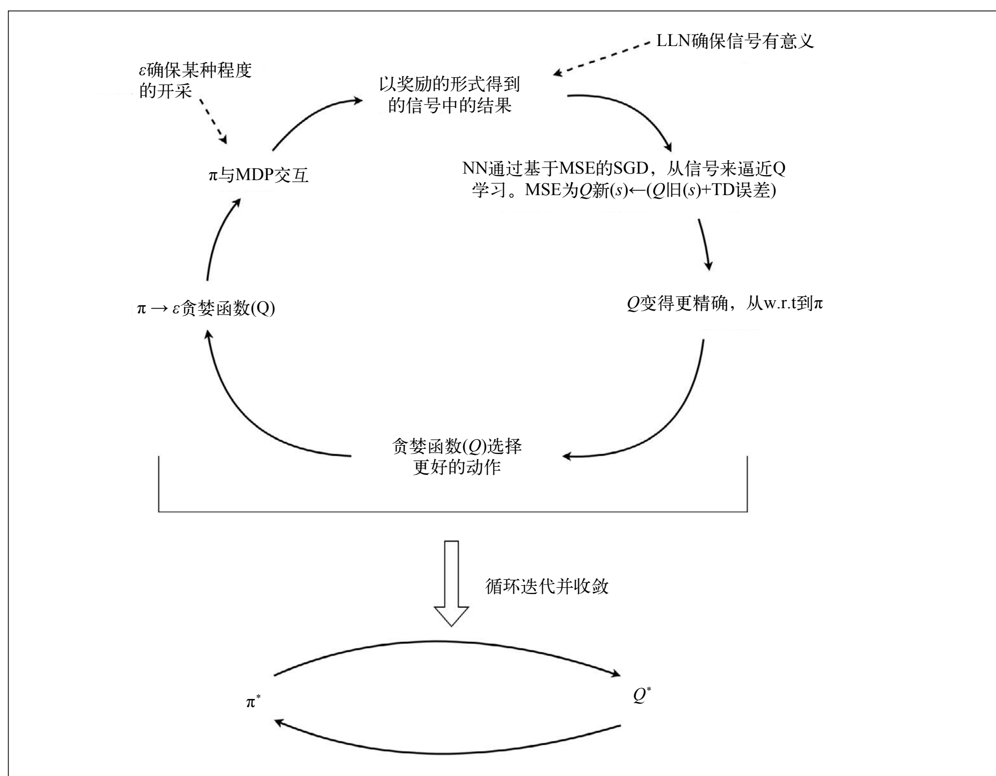


图 B-4：策略迭代

这个算法还要求你以“良好”的方式对状态采样：它应该按比例选取能够代表通常在游戏中的状态（或者至少是目标代理级别的游戏类型）。

B.2.4 贝尔曼方程

Q 方程可以转化为贝尔曼方程，如下所示：

$$
\begin{aligned}
&Q_\pi(s,a)=E\left[r_t+\gamma r_{t+1}+\cdots+\gamma^n r_n \mid s_t=s,a_t=a\right] \\
&=E\left[r_t+\gamma r_{t+1}+V(s_{t+1}) \mid s_t=s,a_t=a\right]=E\left[r_t+\gamma r_{t+1}+\cdots+\gamma \max{}_{a'}Q\ (s_{t+1},a') \right.\\
&\left.\mid s_t=s,a_t=a\right]
\end{aligned}
$$

正如在蒙特卡洛方法中，我们可以做许多 Q 的更新。

MSE：

$$
Q_\pi(s_t,a_t)\leftarrow Q_\pi(s_t,a_t)+\alpha\left[\left(\underbrace{\underbrace{r_t+\max{}_a Q_\pi\left(s_{t+1},a\right)}_{}{}_{\text{target}}-(Q_\pi)s_t,a_t}_{}{}_{\text{TD}-\text{error}}\right)\right]
$$

TD 误差是时间差误差。实际上，我们正在计算 Q 近似在未来期望值加上已实现的奖励与神经网络所评估的当前值之间的差异。

贝尔曼方程仅在某些边界条件下才有意义。如果 s 是终止状态：

$$V(s)=0$$

对于任何 a：

$$Q(s_{t-1},a)=r_t$$

靠近终止状态的状态首先收敛，因为它们在链中更接近“真”标签，即已知的边界条件。在围棋或国际象棋中，我们通过将 +1 分配给导致最终赢棋的移动（−1 代表输棋，0 代表其他）来应用强化学习。它通过在两个极值 [−1; 1] 之间找到一个点来扩散 Q 值。Q 值接近 0 的移动表示会平局，Q 值接近 1 的移动表示几乎会胜利。

令人惊讶的是，这些移动不仅有 −1 和 1 值（因为偏离最佳路径应该是致命的）。计算 Q 值时的有趣的一点是我们认识到在许多游戏 /MDP 中，没有错误本身就是致命的，正是它们的积累才真正杀死了你，AI 充满了人生的教训。此外，预期累积的奖励空间比通常想象的要平滑得多。一种可行的解释是期望总是具有平均效应：期望只不过是以概率为权重的加权平均。此外，$\gamma<1$ 长期的效果并不占有优势。能够直接计算每个移动赢得比赛的概率难道不令人兴奋吗？

只要我们在终止状态附近对足够的移动采样，Q 学习就能够收敛。深度强化学习不可思议的力量在于它能够将其学习从访问状态泛化到未访问状态。它能够理解什么是平局或获胜，即使它以前从未见过，这是因为网络能够基于先前见过的模式来抽象模式并理解动作的强度（例如在识别出敌人时进行射击）。



离线和在线强化学习

要了解更多在线和离线强化学习之间的差异，可参阅 Kofzor 写的好文章（http://bit.ly/2tUyTNF）。

B.2.5 初始状态采样

在单人游戏环境中（如 Atari 游戏），实际上不需要学习在每种情况下都玩得很好（尽管如果做到了，那表明已经达到了很好的泛化水平）。我们只需要学会从策略所遇到的状态有效地发挥作用即可。因此可以从初始状态开始，在当前策略可到达的状态中采样，这使得我们能够直接从代理玩过的情节中取样。

B.2.6 Q学习的实现

在这个阶段，可以设计一个简单的 Q 学习原型。示例 B-2 为相关代码。

示例 B-2 用 Scala 编写的 Q 学习的简单原型

```
def epoch() = {

    //在初始状态空间中取样（常为唯一的状态）
    //(often unique state)
    var state = initState

    //只要状态不是终止状态，就重复玩一个情节，并在每次迁移时执行Q更新
    while(!state.isTerminal) {

        //基于ε-贪婪策略对动作取样
        val action = epsilonGreedyAction(state)

        //与环境交互
        val (nextState, reward) = transition(state, action)

        //Q更新
        update(state, action, reward, nextState)

        state = nextState
    }
}

//如上面解释的Q更新的实现
def update(state: State, action: Action, reward: Double, nextState: State) = {
    val target = reward + maxQ(nextState)
    fit((state, action), target)
}

//ε-贪婪策略的实现
def epsilonGreedyAction(state: State) = {
    if (Random.float() < epsilon)
        randomAction(state)
    else
        maxQAction(state)
```

```
    }

    //获取最大的Q值
    def maxQ(state: State) =
        actionsWithQ(state).maxBy(_._2)._2

    //获取最大Q值的动作
    def maxQAction(state: State) =
        actionsWithQ(state).maxBy(_._2)._1

    //返回从某个状态开始的动作列表和迁移的Q值
    def actionsWithQ(state: State) = {
        val stateActionList = available_actions.map(action => (state, action))
        available_actions.zip(neural_net.output(toTensor(state_action_list)))


    def initState: State
```

B.2.7 对$Q(s,a)$建模

a 不作为带有状态的神经网络的附加输入，状态是唯一包含每个可能动作的 Q 值的输入和输出，这只有在整个情节中可用动作一致时才有意义（否则，神经网络输出层的每个状态都得不同）。多数情况下，通过以动作全集 A 为输出并忽略不可能的动作可以解决这个问题（一些论文将不可能的动作的目标设为 0）。

B.2.8 经验回放

以神经网络为 Q 逼近器有一个问题：迁移是非常相关的，这减少了迁移的总体方差，毕竟它们都是从同一个情节中提取出来的。想象一下，如果必须学习一个没有任何记忆的任务（甚至不是短期的），你总是会根据上一个情节来优化学习。

谷歌的 DeepMind 研究团队使用了经验回放，它指 DQN 中最后 N 个迁移（原论文中 N 为 100 万）的窗口缓冲区，并极大地提高了它们在 Atari 上的表现。不要从上一个迁移进行更新，而应将它存储在经验回放中，并从来自同一经验回放的一批随机采样的迁移进行更新。

`epoch()` 变为：

```
    def epoch() = {

        //在初始状态空间中取样（常为唯一的状态）
        //(often unique state)
        var state = initState

        //只要状态不是终止状态，就重复玩一个情节，并在每次迁移时执行Q更新
        while(!state.isTerminal) {

            //基于ε-贪婪策略对动作取样
            val action = epsilonGreedyAction(state)

            //与环境交互
            val (nextState, reward) = transition(state, action)
```

```
            //存储迁移（经验回放只不过是环形缓冲）
            expReplay.store(state, action, reward, nextState)

            //批量中的Q更新
            updateFromBatch(expReplay.getBatch())

            state = nextState
        }
    }
```

压缩

DL4J 的张量库 ND4J 不支持 uint8 作为一等类别，然而灰度中的像素以这个精度进行编码。为了避免浪费太多内存空间，INDArray 会被压缩为 uint8。

B.2.9 卷积层与图像预处理

卷积层擅长检测图像中的局部模式。对于像素，它被用作减少输入到其真实流形的维度所需的处理器。考虑到适当的流形观测数据，决策变得容易得多。

图像处理

可以直接向神经网络提供 RGB，但是网络也必须学习该附加模式。看来大脑天生就能组合颜色（很幸运！），因此容忍预处理似乎是合理的。

图 B-5 展示了你所看到的。



图 B-5：Doom 游戏的截图

图 B-6 显示了神经网络所看到的。

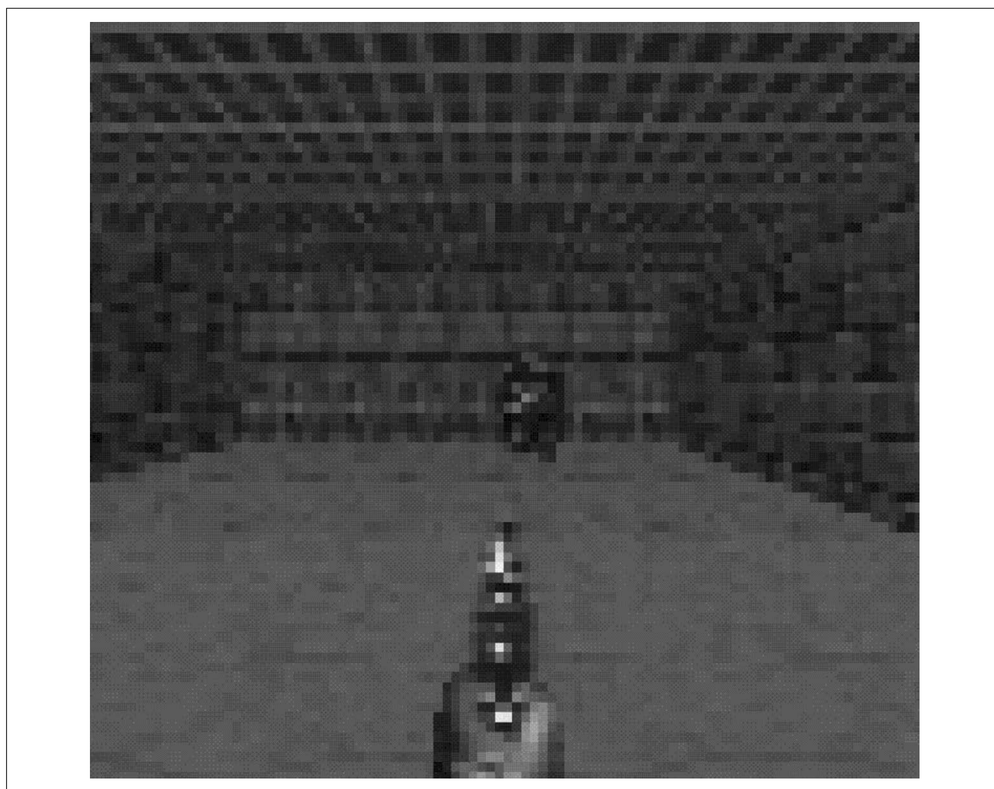


图 B-6：来自同一屏幕截图的神经网络的输入

图像的大小被调整为 84 × 84。随着输入大小的增加，对内存和卷积层计算的需求也随之增加。对于正确地玩游戏来说，图像精微的细节不是必要的。事实上，许多细节纯粹是审美需求，调整到更合理的尺寸会加快训练速度。



跳帧

在最初的 Atari 论文中，四帧中只有一帧是被实际处理的。对于剩余三个图像，重复最后一个动作。它在不丢失太多信息的情况下，使训练速度提高了大约四倍。实际上，Atari 游戏不需要完美地处理每一帧，对于大多数动作来说，保持至少四帧更有意义。

B.2.10　历史处理

为了向神经网络提供关于当前动量的信息，最后四帧（使用跳帧技巧，每四帧中选择一个）被堆叠成四个通道，如图 B-7 所示。这四帧代表一个历史，如 B.1.2 节中讨论的。

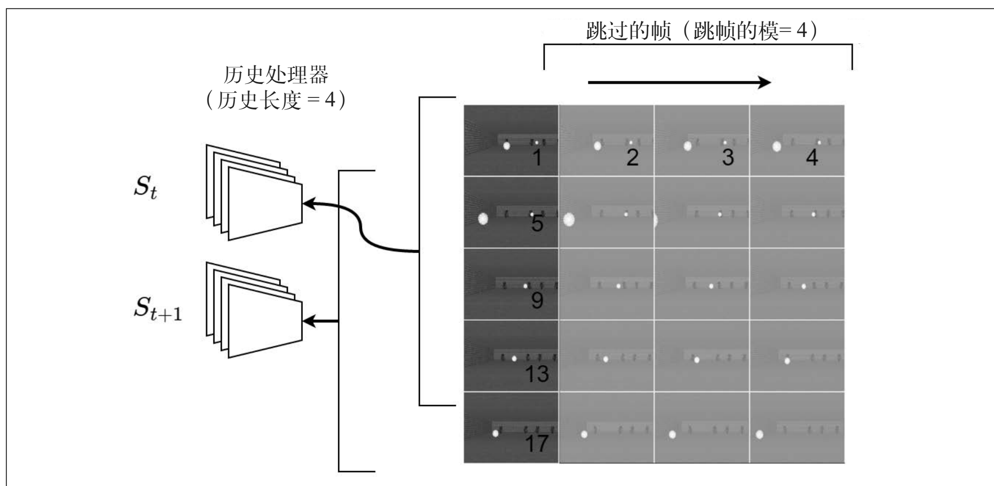


图 B-7：历史处理与堆叠

为了填充历史中的第一批帧，使用随机策略或 noop 回放 [1]。

B.2.11　双Q学习

双 DQN 背后的思想是每 *M* 次更新（硬更新）或每次平滑平均（`target = target × (smooth) + current × (1-smooth)`）更新（软更新）都会冻结网络。的确，它通过在 TD 误差公式中使用不容易"弄坏"的 *Q* 评估来增加学习的稳定性。*Q* 更新公式变为如下所示：

$$Y_{\text{target}} = r_t + \gamma * \left(Q_{\text{target}}\left(s_{t+1}, \arg\max_a Q(s_t + 1, a)\right)\right)$$

B.2.12　裁剪

你可以裁剪 TD 误差（将其绑定在两个限制值之间），这样任何离群值更新都不会对学习产生太大的影响。

B.2.13　缩放奖励

缩放奖励使 *Q* 值较低（在 [-1; 1] 范围内类似于规范化），这样可以显著改变学习效率。这是一个不容忽视的重要超参数。

B.2.14　优先回放

优先回放背后的思想是，并非所有的迁移都是平等的，有些迁移更重要。对它们排序的一种方法是通过它们的 TD 误差。的确，高 TD 误差与高级别的信息相关（令人惊讶）。相比其他样本，应该更频繁地对这些迁移取样。

注 1：为了评估公平，可以使用随机的开始状态。

B.3 图形、可视化和Q均值

为了可视化和调试强化学习的训练或方法，对代理的进展进行可视化监控很有用。这就是为什么我们构建了仪表板 webapp-rl4j（见图 B-8，https://github.com/rubenfiszel/webapp-rl4j）。

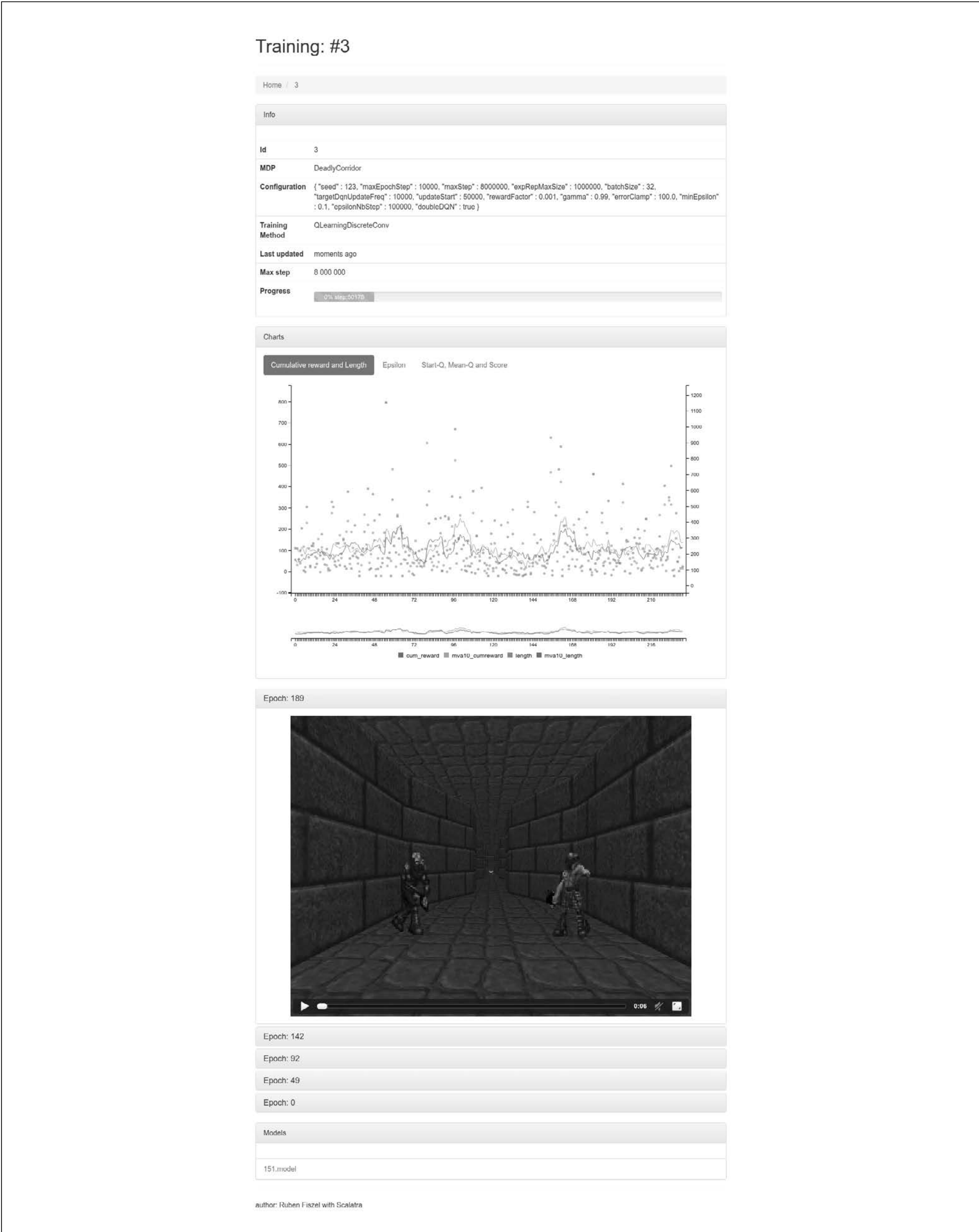


图 B-8：webapp-rl4j 的屏幕截图

最重要的事情是跟踪累积奖励，如图 B-9 所示。这是一种有效地检查代理是否变得更好的方法。需要重点注意的是，它代表 ε- 贪婪策略，而不是直接从 Q 逼近衍生的策略。

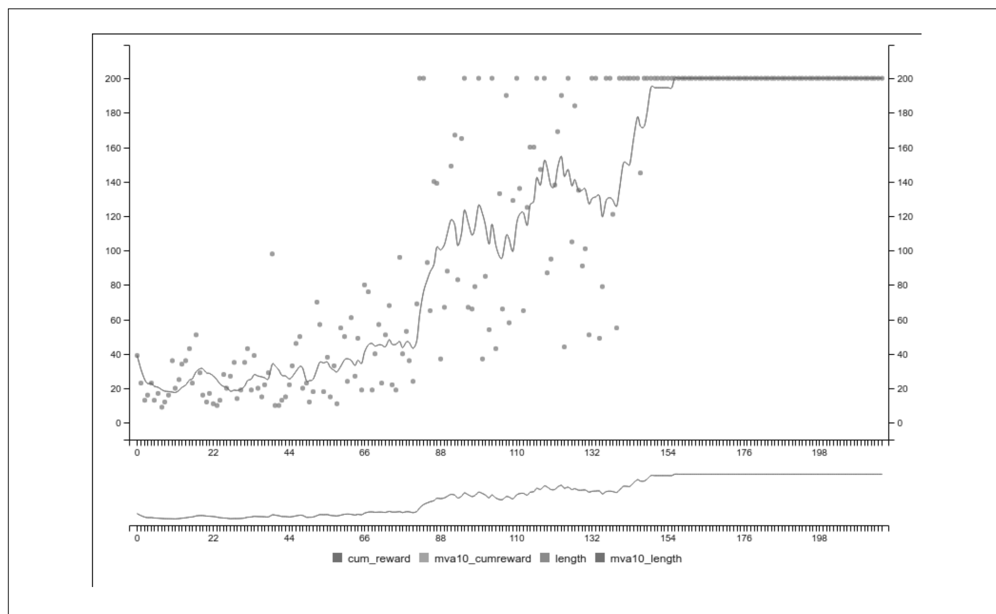


图 B-9：累积奖励图

你可能还想跟踪损失（神经网络的得分）和 Q 均值，如图 B-10 所示。

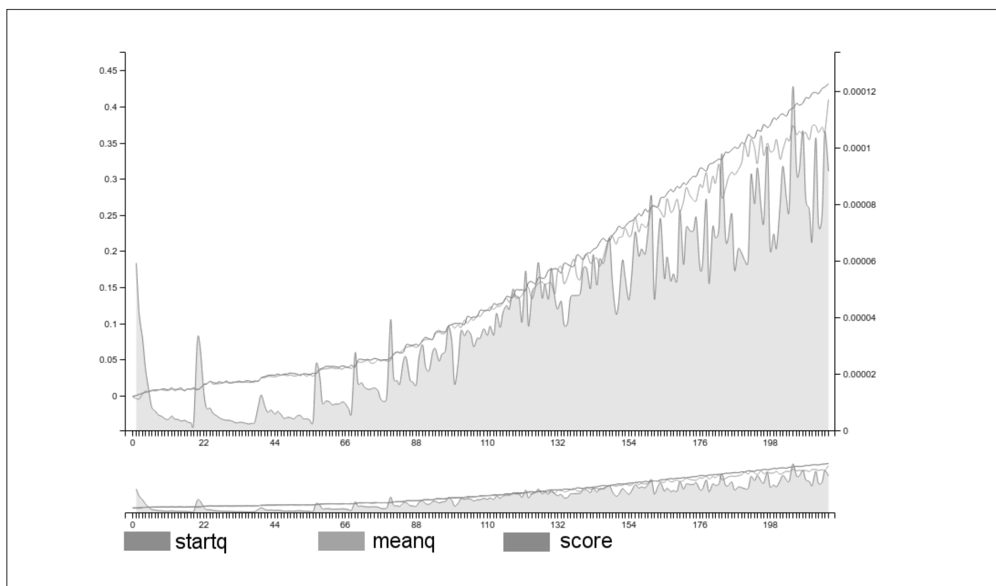


图 B-10：得分和 Q 均值图

与传统的监督学习不同，损失并不一定总是减少，因为学习会影响标签！

如果用于目标网络，你应该看到来自不同目标网络的非连续评估中的一些不连续性。损失应该相对于单个目标网络减少，Q 均值应该平滑地收敛到与平均期望奖励成比例的值。

B.4 RL4J

可以从 GitHub（https://github.com/deeplearning4j/rl4j）上获取 RL4J。目前它实现了 DQN 与经验回放、双 Q 学习和裁剪 [2]，它还包括了带有 A3C 的异步强化学习和异步 N 步 Q 学习。这使得我们可以同时从像素或低维问题（如 Cartpole 游戏）开始。异步强化学习是实验性的，希望大家贡献代码以丰富这个库。

示例 B-3 给出了一个使用 RL4J 中简单的 DQN 来玩 Cartpole 游戏的代码示例，你也可以玩 Doom 游戏。GitHub 上的 rl4j-examples（https://github.com/rubenfiszel/rl4j-examples）包含了更多的例子，你也可以提供自己构造的神经网络模型来试验任何训练方法。

示例 B-3 使用 Scala 开发的基本 RL4J 的代码示例

```
public static QLearning.QLConfiguration CARTPOLE_QL =
        new QLearning.QLConfiguration(
                123, //随机种子
                200, //每轮的最大步骤数
                150000, //最大步骤数
                150000, //经验回放的最大大小
                32, //批量的大小
                500, //目标更新（硬更新）
                10, //noop热身步骤数
                0.01, //奖励缩放
                0.99, //γ
                1.0, //TD误差裁剪
                0.1f, //最小ε
                1000, //ε贪婪退火的步骤数
                true //双DQN
        );

public static DQNFactoryStdDense.Configuration CARTPOLE_NET =
        new DQNFactoryStdDense.Configuration(
                3, //层数
                16, //隐藏节点数
                0.001, //学习率
                0.00 //L2规范化
        );

public static void main( String[] args )
{

    //在新文件夹中记录rl4j-data的训练数据（保存）
    DataManager manager = new DataManager(true);
```

注 2：更多详细信息，请参阅博客原文：https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html。

```
        //从gym(name, render)定义mdp
        GymEnv<Box, Integer, DiscreteSpace> mdp = new GymEnv("CartPole-v0", false,
            false);

        //定义训练
        QLearningDiscreteDense<Box> dql = new QLearningDiscreteDense(mdp,
            CARTPOLE_NET, CARTPOLE_QL, manager);

        //训练
        dql.train();

        //获取最终策略
        DQNPolicy<Box> pol = dql.getPolicy();

        //序列化并保存（序列化展示，但非必需）
        pol.save("/tmp/pol1");

        //关闭mdp（关闭http）
        mdp.close();


    }
```

B.5 小结

这是一段令人兴奋的深度强化学习的旅程。从方程到代码，Q 学习都是一个强大而相对简单的算法。强化学习领域非常活跃且有前景。事实上，可以把监督学习看作强化学习的一个子集（通过设置标签作为奖励）。也许有一天，强化学习将成为人工智能的灵丹妙药。在那之前，我们可以期待它的多样化应用出现在越来越多的令人印象深刻的问题上。最后我要感谢 Skymind 公司及其优秀的团队，感谢这次丰富多彩的实习。

附录C

每个人都需要了解的数字

表 C-1 列出了读者需要了解的数字，也称“Jeff Dean 的 12 个数字”。

表C-1：Jeff Dean的12个数字

计算机架构操作	持续时间（纳秒）
读取 L1 缓存数据	0.5
分支误预测	5
读取 L2 缓存数据	7
互斥锁 / 解锁	100
读取主内存数据	100
使用 Zippy 算法压缩 1KB 数据	10 000
在 1Gbps 的网络上发送 2KB 数据	20 000
从内存顺序读取 1MB 数据	250 000
在同一个数据中心往返	500 000
磁盘搜索	10 000 000
从网络顺序读取 1MB 数据	10 000 000
从磁盘顺序读取 1MB 数据	30 000 000
从加州→荷兰→加州发送数据包	150 000 000

附录D

神经网络和反向传播：数学方法

作者：Alex Black

D.1 介绍

本附录将研究支撑神经网络训练的数学基础：**反向传播算法**。但在深入研究之前，先来思考一个问题：在训练神经网络时，我们想做的是什么？

（先不讨论过拟合等问题）核心问题是，我们希望在训练过程中调整神经网络的参数（基于训练数据)，使网络能够准确预测。这里的两个关键词是**准确的预测**和**调整参数**。

暂时考虑一下分类的任务，对于该任务，给定一些输入数据，要求神经网络预测示例的类别。量化分类预测的好坏程度有很多不同的方法：准确度、F1 评分、负对数似然等。所有这些都是有效的度量，但是其中一些更难优化。比如对网络中的任何给定参数进行微小改动，网络的准确度可能不会改变。因此，使用基于梯度的方法直接优化准确度是不可能的(即“准确度”是不可微的)。

相反，当我们改变参数值时，即使是微小的更改，其他的度量都将增加或减少，如负对数可能性。如果我们限制自己用可微类型的损失函数（如负对数似然）来量化网络预测的质量，并应用一些微积分，就会得到一个简单而优雅的神经网络训练算法。

反向传播算法背后的关键思想很简单。

- 基于可微损失函数量化网络当前预测的好坏程度。
- 通过将多变量微积分的规则应用于损失函数和网络结构,来计算网络中每个参数的梯度。
- 使用计算出的梯度，在使该损失函数最小化的方向上迭代地调整网络参数。

以算法形式对 SGD 算法的说明如下所示：

输入：网络参数 $\boldsymbol{w}$，损失函数 L，训练数据 D，学习率 $\alpha>0$

while 未满足终止条件 **do**

(features, labels) ← D.getRandomMiniBatch()

out ← getNetworkOutput($\boldsymbol{w}$, features)

$\frac{\partial L}{\partial \boldsymbol{w}}$← calculateParameterGradients(w, L, out, labels)

$\boldsymbol{w} \leftarrow \boldsymbol{w} - \alpha \frac{\partial L}{\partial \boldsymbol{w}}$

end

这里的核心部分是计算偏导数$\frac{\partial L}{\partial \boldsymbol{w}}$。如果你不熟悉偏导数，可以把它们看作描述兴趣量（损失函数 L 的值）如何随函数其他量（权重向量 $\boldsymbol{w}$ 中每个参数 w_i 的值）的变化而变化（假设保持其他所有值不变）。



步大小

在大多数情况下，$\frac{\partial L}{\partial \boldsymbol{w}}$也是 w_i 的函数，这就是为什么：(1) 我们只需少数几步（即使用较小的学习率 α）；(2) 必须在修改每次迭代的参数值之后重新计算梯度。

如果 $\frac{\partial L}{\partial w_i}$ 是正数，少量增加 w_i 会使当前样本的损失 L 增加，减少 w_i 会减少损失函数的值。本质上，这就是反向传播算法的全部内容：定义一个损失函数，计算损失函数对所有参数的导数，并使损失函数朝最小化的方向迈出一小步。这是一个简单但非常强大的想法，实际上它是所有深度学习的基础。

D.2　多层感知器中的反向传播

文字介绍得差不多了，下面转到数学表达式。以一个简单的多层感知器为例，即标准的全连接前馈神经网络层，或 DL4J 中的“密集层”，它的平方误差损失函数之和（DL4J 中的 L2 损失函数）的表达式如下所示：

$$L(y,\hat{y})=\sum_{i=1}^{N}(y_i-\hat{y}_i)^2$$

其中 N 是输出的数量，y_i 是第 i 个标签，$\hat{y}_i$=output($\boldsymbol{w}$, $\boldsymbol{f}$) 是在给定特征向量 $\boldsymbol{f}$ 和当前参数 $\boldsymbol{w}$ 的情况下，网络对 y_i 的预测。

图 D-1 展示的是单层神经网络。当你读到本附录剩余部分数学方面的内容时，也可以参考这张图。

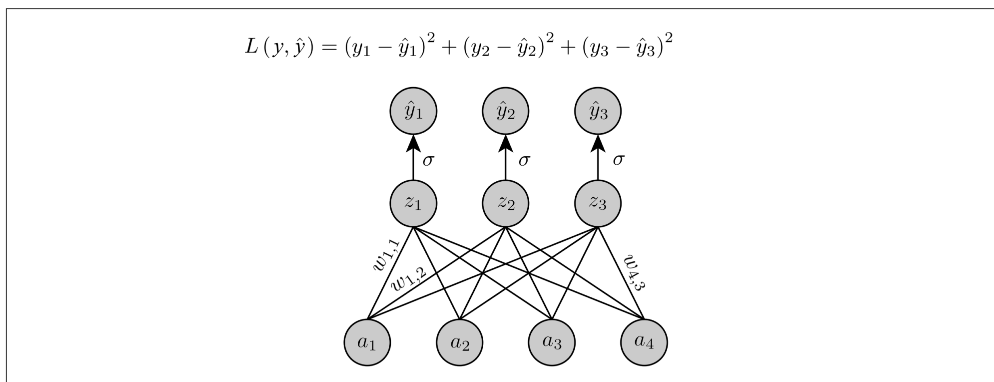


图 D-1：单层神经网络

如果当前层的输入向量是向量 $\boldsymbol{a}$（长度为 4）、元素相关的非线性函数（激活函数，例如 tanh 或 ReLU）为 σ，则该网络的前向传递方程是：

$$z_i = b_i + \sum_{j=1}^{4} w_{j,i} a_i$$
$$\hat{y}_i = \sigma(z_i)$$

其中 b_i 是偏置，而 $w_{j,i}$ 是连接输入 j 和神经元 i 的权重。

给定损失函数，我们要计算的一阶偏导数是关于网络输出 $\hat{y}_i$ 的：

$$\frac{\partial L}{\partial \hat{y}_j} = \frac{\partial}{\partial \hat{y}_j}\left(\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2\right)$$
$$= \frac{\partial}{\partial \hat{y}_j}\left(y_j - \hat{y}_j\right)^2$$
$$= -2\left(y_j - \hat{y}_j\right)$$

随着与网络结构相反的方向，接下来要计算 $\frac{\partial L}{\partial z_i}$ 作为 $\frac{\partial L}{\partial \hat{y}_i}$ 的函数，这将取决于激活函数 $\sigma(z)$ 的数学形式。为了简单起见，我们假定一个简单的激活函数，如 sigmoid、tanh 或 ReLU（因为它们的导数是简单的元素相关的函数，但是一些函数如 softmax 则没有这个特性）。

$$\frac{\partial L}{\partial z_i} = \frac{\partial L}{\partial \hat{y}_i}\frac{\partial \hat{y}_i}{\partial z_i}$$
$$= \sigma'(z_i)\frac{\partial L}{\partial \hat{y}_i}$$

例如在使用 sigmoid 激活函数时，$\sigma(z) = \frac{1}{1+\mathrm{e}^{-z}}$，导数 $\sigma'(z) = \sigma(z)(1-\sigma(z))$ 。

接下来根据给定的先前计算的导数 $\frac{\partial L}{\partial z_i}$ ，我们应用链式法则计算权重 $w_{j,i}$ 的偏导数：

$$
\begin{aligned}
\frac{\partial L}{\partial w_{j,i}} &= \sum_{k=1}^{3} \frac{\partial L}{\partial z_k} \frac{\partial z_k}{\partial w_{j,k}} \\
&= \frac{\partial L}{\partial z_i} \frac{\partial z_i}{\partial w_{j,i}} \\
&= \frac{\partial L}{\partial z_i} \frac{\partial}{\partial w_{j,i}} \left(b_i + \sum_{k=1}^{4} w_{k,i} a_i \right) \\
&= a_i \frac{\partial L}{\partial z_i}
\end{aligned}
$$

可以用同样的方法来确定偏置的导数，表达式为：$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial z_i}$。

最后计算输入激活值 a_i 的损失函数的导数，再次根据给定的 $\frac{\partial L}{\partial z_i}$：

$$
\begin{aligned}
\frac{\partial L}{\partial a_i} &= \sum_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial z_j}{\partial a_i} \\
&= \sum_{j=1}^{3} \frac{\partial L}{\partial z_j} \frac{\partial}{\partial a_i} \left(b_j + \sum_{k=1}^{4} w_{k,j} a_j \right) \\
&= \sum_{j=1}^{3} \frac{\partial L}{\partial z_j} w_{i,j}
\end{aligned}
$$

这样就行了。如果下面还有一层，可以采用完全相同的方法，但是要用 $\hat{y}_i$。我们只是随着与网络结构相反的方向应用链式规则。

将这个分析扩展到小批量（即多个样本）的情况非常简单：可以简单地平均每个参数的梯度，而不是平均每个小批量的样本。实际上，这些方程是在小批量数据上使用矩阵乘法和向量运算实现的。此外，可以使用相同的方法计算其他（更复杂的）模型的导数，如 RNN 和 CNN。

附录E

使用ND4J API

ND4J 是在 JVM 上运行的一个科学计算库。它被设计为能够在生产环境中快速运行，其主要特点如下所示：

- 多用途 n 维数组对象；
- 支持包括 GPU 在内的多平台；
- 线性代数和信号处理功能。



ND4S 是 ND4J 的 Scala 版本。

可用性的鸿沟使 Java、Scala 和 Clojure 程序员与数据分析领域最强大的工具，比如 NumPy 或 Matlab 绝缘。像 Breeze 这样的库不支持 n 维数组或张量，而这些对于深度学习和其他任务是必需的。ND4J 和 ND4S 被美国国家实验室用于气候建模等任务，这些任务需要计算密集的模拟。

ND4J 将 Python 社区直观的科学计算工具带到了 JVM 世界，它开源、分布式，并且支持 GPU 库，结构类似于 SLF4J。ND4J 为工程师们提供了一种简单的方法来移植他们的算法，以及与 Java 和 Scala 生态系统中其他库的接口到生产环境。



完整的在线 ND4J 用户指南

ND4J 支持的操作不限于本附录中所列出的。访问 http://nd4j.org/userguide 查看完整的 ND4J 用户指南。



完整的 ND4J API Javadoc

访问 http://nd4j.org/doc 中的 Javadoc 查看完整的 ND4J API 资源。

E.1 设计与基本用法

ND4J 被设计为可在许多环境中运行，并与如今的现代数据系统集成。以下是它的一些特点。

- 通过 CUDA 支持 GPU。
- 与 Hadoop 和 Spark 的集成。
- 模仿 NumPy 的语法设计的 API。

ND4J 中的大多数操作都专注于处理数字数组，其核心数据结构被称为 NDArray。

E.1.1 理解NDArray

NDArray 本质上是一个具有一定维度的矩形数组。可以通过以下属性量化 NDArray。

- 秩
- 形状
- 长度
- 步长
- 数据类型

接下来解释其中每个属性对于 NDArray 的作用。



NDArray 与 INDArray

我们使用 NDArray 这个术语来代表一个 *n* 维数组的一般概念。术语 INDArray 具体指由 ND4J 定义的 Java 接口，在实践中这两个术语可以互换使用。

- ❑ 秩

  NDArray 的秩是维度的数量。二维 NDArray 的秩为 2，三维数组的秩为 3，以此类推。可以创建任意秩的 NDArray。

- ❑ 形状

  NDArray 的形状定义了每个维度的大小。假设有一个 3 行 5 列的二维数组，那么这个 NDArray 的形状为 [3, 5]。

- ❑ 长度

  NDArray 的长度定义了数组中元素的总数。长度总是等于构成形状的值的乘积。

- ❑ 步长

  NDArray 的步长指每个维度中相邻元素的间隔（在底层数据缓冲区中）。步长是在每个

维度分别定义的，所以秩为 *N* 的 NDArray 具有 *N* 个步长值，每个维度一个。请注意，大多数时候，你不需要知道（或者关心）步长——你只需要知道这是 ND4J 内部的操作方式即可。下一节有一个关于步长的例子。

❑ 数据类型

NDArray 的数据类型指它所包含的数据的类型（例如浮点或双精度）。注意，在 ND4J 中这个值是全局设置的，因此所有 NDArray 的数据类型都相同。本文稍后将讨论数据类型的设置。



需要了解的关于 NDArray 索引和维度的事情

行的维度为 0，列的维度为 1，因此，`INDArray.size(0)` 是行数，`INDArray.size(1)` 是列数。与大多数编程语言中的普通数组一样，索引是从 0 开始的，因此行的索引范围为 `0` 到 `INDArray.size(0)-1`，其他维度也是一样。

NDArray 与物理内存存储

NDArray 作为一个单一扁平的数字数组（或者通常作为单个连续的内存块）存储在内存中，因此与传统的 Java 多维数组有很大的不同，如 `float[][]` 或 `double[][]`。

在物理上，INDArray 底层的数据是堆外存储的，即它存储在 JVM 之外。这样做有许多好处，包括性能、与高性能 BLAS 库的互操作性，以及能够避免 JVM 在高性能计算中的某些缺点（例如 Java 数组由于整数索引的上限而被限制为 $2^{31}-1$（21.4 亿）个元素的问题）。

E.1.2 ND4J一般的语法

ND4J 中使用的操作有三种：

- 标量
- 转换
- 累积

大多数操作只接受枚举值，或者可以自动完成的离散值列表。标量、转换和累积都有各自的模式，下面分别了解它们。

❑ 标量

标量只接受两个参数：输入和要应用于该输入的标量。例如 `ScalarAdd()` 接受两个参数：输入 `INDArray x` 和标量 `Number num`，即 `ScalarAdd(INDArray x, Number num)`。同样的格式适用于每个标量操作。

❑ 转换

转换是最简单的，因为它们接受单个参数并对其执行操作。绝对值是接受参数 `x`（如 `abs(IComplexNDArray ndarray)`）并产生结果的转换，该结果为 `x` 的绝对值。类似地，可以应用 sigmoid 转换函数 `sigmoid()`，以产生“x 的 sigmoid”。

- 累积

  最后还有累积操作，也就是 GPU 上下文中的降维。累积将数组和向量彼此相加，并且可以通过行操作将它们的元素相加，来降低结果中那些数组的维度。例如对以下数组执行累积：

```
[1 2
 3 4]
```

所得向量如下所示：

```
[3
 7]
```

这将列（即维度）从两个减少到一个。

累加可以是成对的，也可以降维为标量。在成对降维中，可能要处理两个形状相同的数组：*x* 和 *y*。在这种情况下，可以通过两个两个地取 *x* 和 *y* 的元素，来计算它们的余弦相似度。

```
cosineSim(x[i], y[i])
```

或者，计算 `EuclideanDistance(arr, arr2)`，得到一个数组 `arr` 和另一个数组 `arr2` 之间的降维值。

E.1.3　使用NDArray的基础知识

1. ND4J类

这个类有各种静态帮助方法来协助创建 NDArray。下面将讨论 NDArray 一些更常用的方法。

`Nd4j.zeros(int...)`。数组的形状被指定为整数，例如创建一个包含 3 行和 5 列的零填充数组，使用 `Nd4j.zeros(3,5)`。

`Nd4j.ones(int...)`。此方法与 `.zeros(int...)` 类似，但是用 1 而不是 0 填充返回的 NDarray。

用其他值初始化。可以将 ND4J 类方法与其他操作相结合，创建具有其他值的数组。例如要创建一个用 10 填充的数组，可以这样做：

```
INDArray tens = Nd4j.zeros(3,5).addi(10)
```

这个初始化过程分为两步：首先分配一个零填充的 3 × 5 数组，然后每个值加 10。

用随机数初始化。ND4J 提供了几种生成元素值为伪随机数的 INDArray 的方法。

要产生在 0 至 1 范围内的均匀随机数，使用：

```
Nd4j.rand(int nRows, int nCols)
```

(生成二维数组)，或：

```
Nd4j.rand(int[])
```

(生成三维或更高维度)。

类似地，生成平均值为零和标准差为 1 的高斯随机数，可以使用：

```
Nd4j.randn(int nRows, int nCols)
```

或

```
Nd4j.randn(int[])
```

考虑到重复性（即，设置 ND4J 的随机数生成器种子），可以这样做：

```
Nd4j.getRandom().setSeed(long)
```

2. 控制NDArray的形状

接下来的小节讨论在执行基本操作的过程中如何控制 NDArray 的形状。

3. 创建基本数组

下面的示例创建了一个具有两列的一维数组，列的值为 {1, 2}。

```
INDArray nd = Nd4j.create(new float[]{1,2},new int[]{2}); //向量行
```

示例：创建 2×2ND Array。下面的示例创建了一个有两行和两列的二维 NDArray。第一行的值为 {1, 2}，第二行的值为 {3, 4}。

```
INDArray arr1 = Nd4j.create(new float[]{1,2,3,4},new int[]{2,2});
System.out.println(arr1);
```

输出如下所示：

```
[[1.0 ,3.0]
[2.0 ,4.0]
]
```



注意 ND4J 中行和列的排列

可以以 C（行为主序）或 Fortran（列为主序）的序列方式对 NDArray 编码。有关行与列为主序的详细信息，请参阅 https://en.wikipedia.org/wiki/Row-major_order。

ND4J 可以同时使用 C 和 Fortran 数组序列方式的组合。大多数用户可以只使用默认的数组序列，但是请注意，如果需要，可以对给定的数组使用特定的序列方式。

示例：将两个 2×2ND Array 相加。创建第二个数组（`arr2`）并将其加到第一个数组（`arr1`）。

```
INDArray arr2 = ND4j.create(new float[]{5,6,7,8},new int[]{2,2});
arr1.addi(arr2);
System.out.println(arr1);
```

输出如下所示：

```
[[7.0 ,11.0]
[9.0 ,13.0]]
```

4. 从Java数组创建NDArray

ND4J 为从 Java 的 float 和 double 数组创建 NDArray 提供了方便的方法。

要从一维 Java 数组创建一维 NDArray，方法如下所示。

- 行向量：`Nd4j.create(float[])` 或 `Nd4j.create(double[])`
- 列向量：`Nd4j.create(float[],new int[]{length,1})` 或 `Nd4j.create(double[],new int[]{length,1})`

对于二维数组，使用 `Nd4j.create(float[][])` 或 `Nd4j.create(double[][])`。

从具有三个或更多维度（`double[][][]` 等）的 Java 原始类型数组创建 NDArray，一种方法如下：

```
double[] flat = ArrayUtil.flattenDoubleArray(myDoubleArray);
int[] shape = ...; //这里为数组形状
INDArray myArr = Nd4j.create(flat,shape,'c');
```

5. 获取和设置不同的NDArray值

对于 INDArray，可以通过你想去获取或设置的元素的索引来获取或设置值。秩为 *N* 的数组（即具有 *N* 维的数组），需要 *N* 个索引。



优化 NDArray 操作的性能

从性能方面考虑，单独获取或设置值（例如在 `for` 循环中一次处理一个值）通常不好。如果可能，尝试使用一次性对大量元素进行操作的其他 INDArray 方法。

要从二维数组中获取值，方法如下：

```
INDArray.getDouble(int row, int column)
```

对于任意维度的数组，方法如下：

```
INDArray.getDouble(int...)
```

要获得索引 `i`、`j`、`k` 的值，方法如下：

```
INDArray.getDouble(i,j,k)
```

若要设置值，可使用下面任一种 `putScalar` 方法。

- `INDArray.putScalar(int[],double)`
- `INDArray.putScalar(int[],float)`
- `INDArray.putScalar(int[],int)`

这里，`int[]` 是索引，而 `double/float/int` 是要放在那个索引上的值。

6. 使用NDArray行

处理 NDArray 的部分信息有多个帮助方法。

获取一行。若要从 INDArray 中获取一行，方法如下：

```
INDArray.getRow(int)
```

这显然会返回一个行向量。这里需要注意：这行是一个视图，对返回行的更改将影响原始数组，这个特性有时非常有用（例如 `myArr.getRow(3).addi(1.0)` 将 1.0 添加到一个大数组的第三行）。要复制行，可以使用：

```
getRow(int).dup()
```

获取多行。类似地，要获得多行，方法如下：

```
INDArray.getRows(int...)
```

这将返回一个行堆叠的数组。但是请注意，这将是原始行的副本（不是视图），由于 NDArray 在内存中的存储方式，因此这里不可能有视图。

设置单行。若要设置单个行，方法如下：

```
myArray.putRow(int rowIdx,INDArray row)
```

这将 `myArray` 的第 `rowIdx` 行设置为值包含在 INDArray row 中。

7. 确定NDArray的大小/维度的快速参考

以下方法由 INDArray 接口定义。

- 获取维度的数量：`rank()`。
- 仅用于二维 NDArray：`rows()`、`columns()`。
- 第 *i* 个维度的大小：`size(i)`。
- 获取所有维度的大小，结果为 `int[]`：`shape()`。
- 确定数组中元素的总数：`arr.length()`。
- 参见：`isMatrix()`、`isVector()`、`isRowVector()` 和 `isColumnVector()`。

E.1.4 Dataset

`org.nd4j.linalg.dataset.DataSet` 类表示具有输入和输出数据的数据转换（http://nd4j.org/doc/org/nd4j/linalg/dataset/DataSet.html）。

在神经网络的场景中，Dataset 表示输入特征到输出向量（如结果）的映射。结果专门用于神经网络编码，使得任何被视为真的标签都是 1，其余的都是 0。

1. 与NDArray的关系

NDArray 表示输入和输出的特定数据向量，`DataSet` 对象表示输入 NDArray 和输出 NDArray 的组合，这就是表示学习过程的函数输入和输出的方式。

2. 常见用法

下面的代码示例展示如何使用 ND4J API 从 `DataSet` 类实例获取特征和标签，以及这些结构如何传递给经过训练的模型并生成结果 NDArray。

```
DataSet t = ...
INDArray features = t.getFeatureMatrix();
INDArray lables = t.getLabels();
INDArray predicted = model.output(features,false);
```

这个简单的代码片段说明了 DataSet、NDArray 和 DL4J 模型的核心关系。

E.2 创建输入向量

之所以强调向量创建，是因为它是每个建模工作使用的核心技术。你需要了解向量化策略，然后应用 ND4J API 实现向量化策略。正如前面提到的，建模输入数据是与输出向量相关的一组输入特征。本节讲解如何创建它们并将其与 `DataSet` 对象绑定。



向量化的简单示例

这些代码片段旨在简单展示设置特征和标签值的基本概念，在实践中，可能有其他更高效的 ND4J 方法。很多时候，DL4J 记录读取器将为实践者手动完成大部分工作，本节只是展示如何实际地操作数据。

向量创建的基本方法

先创建一个简单的有两列的特征向量，如下所示：

```
INDArray myFeatures = ND4j.create(new float[]{0.5, 0.5},new int[]{1,2});
```

1. 设置向量的大小

使用 `ND4j.create()` 方法的第二个参数控制向量的大小。

```
new int[]{1,2}
```

该参数指示 ND4J 创建一个具有两个特征列的单行 NDArray。

2. 设置特征值

创建特征值的方法有很多种，下面的代码片段展示了在向量中手动设置特征值最简单的方法。

```
for ( int row = 0; row < myFeatures.rows(); row++ ) {
    for ( int col = 0; col < myFeatures.getRow( row ).columns(); col++ ) {
        myFeatures.getRow(row).putScalar(col, 0.9);
    }
}
```

在这个示例中，每行中每一列的值设置为 0.9。

3. 设置标签

设置标签与设置特征值一样简单，因为输入特征数据和输出数据使用相同的数据结构：NDArray。为了建立标签数据，需要根据正在建立的模型类型以及我们要求的输出策略对数据进行编码。

单标签输出。在这种情况下，用于标签的输出 NDArray 将具有单列，我们将一个标签的值设置为 0.0，将另一个标签的值设置为 1.0。

```
myFeatures.getRow(0).putScalar(0, 1.0);
```

多标签输出。在这种情况下，用于标签的输出 NDArray 中每个标签都有一列。训练数据集中的每个类别（标签）将匹配输出 NDArray 中的特定列。基于训练的目的，我们将与标签

索引相同的列的值设置为 1.0，如下面的示例所示：

```
myFeatures.getRow(0).putScalar(0, 0.0);
myFeatures.getRow(0).putScalar(1, 1.0);
myFeatures.getRow(0).putScalar(2, 0.0);
```

回归的输出。在这种情况下，用于回归模型输出值的输出 NDArray 将具有单列。我们将这个值设置为与输入向量关联的浮点值。

```
myFeatures.getRow(0).putScalar(0, 55.25);
```

E.3　使用MLLibUtil

DL4J 平台还包含支持与其他机器学习库互操作的工具。本节将介绍 `MLLibUtil` 类的一些方法，以帮助处理 Spark 向量对象。

E.3.1　从INDArray转换到MLLib向量

如果需要将 NDArray 转换为 MLLib 向量，可以使用 `MLLibUtil.toVector( INDArray )` 方法，如下所示：

```
Vector prediction = MLLibUtil.toVector( myNDArray );
```

E.3.2　从MLLib向量转换到INDArray

如果需要将 MLLib 向量转换为 NDArray 向量，可以使用 `MLLibUtil.toVector(Vector)` 方法，如下所示：

```
INDArray ndArray = MLLibUtil.toVector( labeledPoint.features() );
```

E.4　使用DL4J进行模型预测

本节研究 DL4J 和 ND4J 中类的交互，以便在 API 级别进行模型预测。

同时使用DL4J和ND4J

通常在由 DL4J 中的模型做出预测的场景中，使用 `MultiLayerNetwork` 方法来表示模型，并将 INDArray 对象作为输入传递到 `output()` 方法。该方法的输出是表示每个标签的概率的列表，其中最大的概率被视为输入的“预测标签”。

```
MultiLayerNetwork.output(INDArray input, boolean train)
```

稍后更多示例将展示使用这个方法与经过训练的 DL4J 模型，来产生由 ND4J NDArray 表示的输出。



输出值来自哪里？

这些值是基于输入向量的前馈传递之后网络最终层（“输出层”）的激活值。

由 `output()` 方法返回的包含在 NDArray 中的输出值通常如下所示：

```
[0.5, 0.5]
```

这个例子中有一个一行两列的 NDArray，每一列表示与其相关联的标签的得分，每行的列值的含义取决于生成的输出层的类型。下面将展示如何处理由不同输出层生成的不同类型的输出。

1. 不同类型输出层的输出向量的区别

记住，softmax 激活函数限制输出的总和为 1.0（如概率），而 sigmoid 激活函数则分别限制输出值。例如 softmax 层输出的概率加起来会达到 1.0，但是 sigmoid 层输出的每个单元可能都是 0.9，sigmoid 层不具有这种“横向”约束。

用于二元分类的逻辑输出层。这种类型的层的输出表示二元标签为真的概率。

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
```

输出向量中每个条目表示特定类型的标签，并且它们的概率值与其他标签不相关。

用于多标签分类的 softmax 输出层。softmax 层的输出将是一个概率列表，总和为 1.0。

```
INDArray predictions = trainedNetwork.output( ndArrayFeatures );
for ( int row = 0; row < output.rows(); row++ ) {
    System.out.println( "Input Row: " + row );
    for ( int col = 0; col < output.getRow( row ).columns(); col++ ) {
        System.out.println( "\tColumn: " + col + ":" + output.getRow( row )
            .getDouble( col ) );
    }
}
```

它会给出类似于下面的输出。

```
Input Row: 0
    Column: 0:0.996791422367096
    Column: 1:0.0032307980582118034
Input Row: 1
    Column: 0:0.0016306628240272403
    Column: 1:0.9983481764793396
Input Row: 2
    Column: 0:0.0016311598010361195
    Column: 1:0.9983481168746948
Input Row: 3
    Column: 0:0.9988488554954529
    Column: 1:0.0011729325633496046
```

输出向量的每个条目表示关联标签的概率，行中的每列与特定的标签相关联。

用于回归输出的线性输出层。回归模型使用单个输出，该输出表示我们希望建模的“恒等”激活函数（如线性）的输出值。

2. 从返回的INDArray中获取预测标签

我们在已训练的网络上调用 `output()` 方法，获得结果概率，并找到最大概率，如下所示：

```
INDArray predictions = trainedNetwork.output( ndArray );
int maxLabelIndex = Nd4j.getBlasWrapper().iamax( predictions );
```



`.output()` 方法相对于 `.predict()` 的优点

可以在回归和分类时使用 `output()` 方法，因为它用起来稍微灵活一些。

附录F

使用DataVec

作者：Alex Black

DataVec 是一个处理机器学习数据的库，处理机器学习流水线的提取、转换和加载（ETL）或**向量化**组件，其目标是简化原始数据的准备和加载，使其成为便于机器学习使用的格式。DataVec 的功能包括加载用于单机和分布式（Apache Spark）应用程序的表格形式数据（CSV 文件等）、图像和时间序列数据集。



ND4J 向量创建与 DataVec

DataVec 用于处理本书前面提到的许多特征和标签创建的繁杂工作。使用 DataVec 被视作 DL4J 工作流在单机和 Spark 上的最佳实践。

DataVec 主要有两种功能。

- 从多种格式加载数据。
- 执行常见的数据转换操作（通常被称为 data wrangling 或 data munging）。

下面分别介绍这两种功能。

F.1　为机器学习加载数据

机器学习的数据格式多样，有不同的要求以及加载每种格式的数据的库。通常机器学习实践者最终会编写一次性代码来加载数据，这既耗时又易出错。DataVec 通过两种方式缓解这些问题：第一，为常见用例提供数据加载功能，例如读取图像和 CSV 数据；第二，提供用于添加新数据格式或数据源的简单抽象集。



ETL、预处理与向量化

推荐的最佳实践是使用 DataVec 这样的工具将数据预处理成 DL4J 可以轻松读取和自动向量化的格式。手工向量化原始数据是一项劳动密集型活动，你应该尽量避免这么做。

DataVec 提供的抽象集相对简单。

`Writable` 是表示数据的接口。例如可以使用 `DoubleWritable` 表示双精度数值，还可以使用 Text 实例表示文本数据[1]。

`RecordReader` 接口提供了从原始数据格式转换为常见样本格式的机制。具体说来，`RecordReader` 获取原始数据并将其转换为 `List<Writable>` 表示，然后 DL4J 的 `RecordReaderDataSetIterator` 类可以读取该表示，还可以将样本组合为小批量 `DataSet` 对象，如图 F-1 所示。

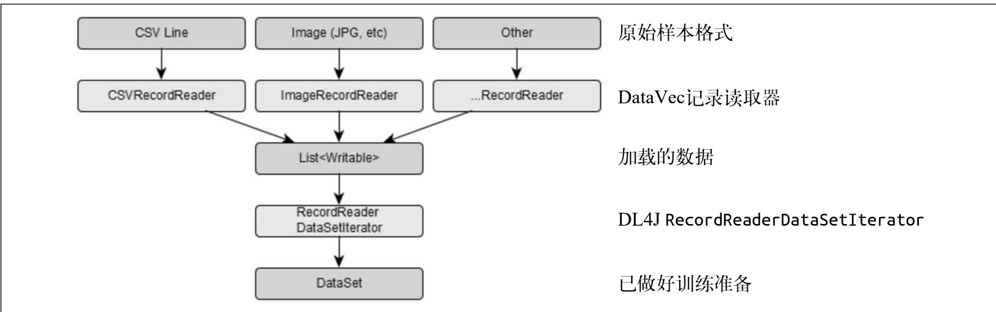


图 F-1：DataVec 处理流水线

类似地，`SequenceRecordReader` 接口提供了加载序列（时间序列）数据的机制。在 DataVec 中，单个（非序列）样本表示为 `List<Writable>`，而序列样本表示为 `List<List<Writable>>`，可以将其看作时间步的列表，即外部列表是时间步的列表，而内部列表是每个时间步的值的列表。换句话说，代码 `mySequence.get(i).get(j)` 将返回第 *i* 个时间步中第 *j* 个值。使用序列记录读取器实际上与使用普通的记录读取器相同，如图 F-2 所示。

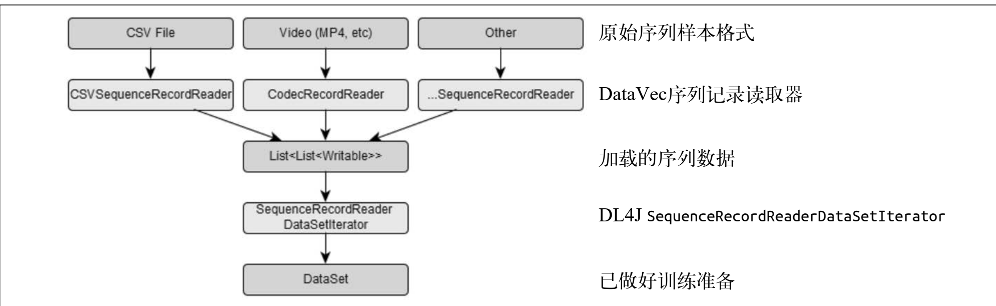


图 F-2：DataVec 序列数据处理流水线

注 1：其他类型包括 `IntWritable`、`LongWritable`、`FloatWritable` 和 `NullWritable`。DataVec 还提供了一个 `NDArrayWritable` 类，用于通过 ND4J 高效地处理数字数组数据，例如图像。

DataVec 和 DL4J 处理训练数据的方法的一个重要特性是，它们只有在需要时，才使用迭代器模式加载数据。这意味着只有在需要时才能加载训练数据（如果需要的话，异步预取），而不是一次性加载所有数据到内存中（这对于大型数据集通常是不可行的）。因为这个原因，`RecordReader`、`SequenceRecordReader` 和 `DataSetIterator` 接口都提供了 `next()`、`hasNext()` 和 `reset()` 方法。

F.2　为多层感知器加载CSV数据

可以使用几行代码加载 CSV 数据，这个过程也可用于其他类型的分隔数据，如制表符分隔格式。

示例 F-1 展示了如何加载 CSV 数据文件并创建 `DataSetIterator`，为在 DL4J 中训练网络做好准备。

示例 F-1　加载 CSV 数据

```
//首先指定文件路径，以及一些属性：

File file = new File( "/path/to/my/file.csv" );

int numLinesToSkip = 0;        //可选项，是否允许跳过标题行

String delimiter = ",";        //逗号分隔


//创建记录读取器，并初始化它：

RecordReader reader = new CSVRecordReader( numLinesToSkip, delimiter );
InputSplit inputSplit = new FileInputSplit( file );
reader.initialize( inputSplit );


//创建DataSetIterator。假设这里要进行分类：

int minibatchSize = 10;        //每个小批量中的样本数

int labelIndex = 7;            //包含标签的列的索引

int numClasses = 5;            //类别数量（标签分类）

DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );


//使用DataSetIterator训练网络：

myNetwork.fit( iterator );
```

需要注意的事项如下。

- 无参数(no-arg)构造函数(new CSVRecordReader())默认为逗号分隔格式,并且跳过 0 行。
- 在 RecordReaderDataSetIterator 中，用户可以自由配置小批量的大小。
- RecordReaderDataSetIterator 中存在许多其他构造函数，用于多标签回归，以及像无监督学习这样的无标签情况。
- 对于分类，RecordReaderDataSetIterator 假定其中一列包含类别索引的整数值，即一列包含在 0 至 numClasses – 1 范围内的整数值。
- 可以使用其他的分隔符。例如 \t 表示以制表符分隔的记录，或者使用静态字段 CSVRecordReader.QUOTE_HANDLING_DELIMITER 处理引号间包含逗号的记录。
- CSVRecordReader 以样本出现在文件中的顺序进行输出。如果数据是非随机顺序的（例如所有类别为0的样本后面跟着所有类别为1的样本),那么应该先增加一个混洗的步骤。

F.3 为CNN加载图像数据

DataVec 的 ImageRecordReader 是用于读取图像数据并执行常见的图像操作（如裁剪）的记录读取器。ImageRecordReader 支持各种图像格式（包括 JPG、PNG、TIFF 和 BMP 等），以及 JavaCV 和 OpenCV 提供的高效加载和图像操作。

ImageRecordReader 的实现在数据文件加载方式上很灵活。示例 F-2 简单演示了图像加载。在这个示例中，假定图像文件在目录中，目录名称是标签（图像类别 / 分类）。例如文件在目录 .../root_directory/label_0、.../root_directory/label_1/ 中。注意，作为标签的目录名称本身可以是任意值。

示例 F-2　加载图像数据

```
//首先指定输入图像深度（通道数）
int inputNumChannels = 3;    //3为RGB彩色图像，1为灰度图像
int outputHeight = 32;       //缩放到32个像素的高度输出
int outputWidth = 32;        //缩放到32个像素的宽度输出

//第二步，指定根目录，并设置允许的文件格式
//我们会使用Random实例来随机打乱顺序
File rootDir = new File( "/path/to/my/root_directory/" );
String[] allowedExtensions = BaseImageLoader.ALLOWED_FORMATS;
Random rng = new Random();
FileSplit inputSplit = new FileSplit( rootDir, allowedExtensions, rng );

//每个图像都关联一个标签，我们使用ParentPathLabelGenerator，把每个文件的路径作为标签
ParentPathLabelGenerator labelMaker = new ParentPathLabelGenerator();
```

```
//创建和初始化ImageRecordReader

ImageRecordReader reader =
    new ImageRecordReader( outputHeight, outputWidth, inputNumChannels, labelMaker );

reader.initialize( inputSplit );


//最后，创建DataSetIterator：

int minibatchSize = 10;        //每个小批量中的样本数

int labelIndex = 1;            //总是为ImageRecordReader设置值为1

int numClasses = 3;            //类别数量（标签分类）

DataSetIterator iterator =
    new RecordReaderDataSetIterator( reader, minibatchSize, labelIndex, numClasses );


//使用DataSetIterator训练网络：

myNetwork.fit( iterator );
```

ImageRecordReader 还支持许多其他功能。

例如为了在图像上实现训练 / 测试分片，可以将示例 F-3 中的代码添加到示例 F-2 的代码中。

示例 F-3　创建测试 / 训练分片

```
InputSplit[] trainTest = inputSplit.sample( null, 80, 20 );

InputSplit trainData = trainTest[0];        //80%训练

InputSplit testData = trainTest[1];         //20%测试


(other code omitted)


reader.initialize( trainData );
```

类似地，为了向流程添加一些额外的转换步骤，可以将 ImageTransform 类传递给 ImageRecordReader 初始化，如示例 F-4 所示，任意翻转和裁剪图像。

示例 F-4　进行转换的步骤

```
int maxCropPixels = 20;

ImageTransform transform =
    new MultiImageTransform( new Random(),
        new FlipImageTransform(), new CropImageTransform( maxCropPixels ) );


reader.initialize( trainData, transform );
```

F.4 为RNN加载序列数据

序列（时间序列）数据有许多可能的表示和格式，本节重点介绍一个简单但有用的格式。

- CSV 数据，每个文件包含一个时间序列（不同文件的时间序列长度可能不同）。
- 在 CSV 文件中，每行代表一个时间步。
- 特征和标签出现在所有时间步中，并且每个样本的特征和标签都包含在同一个文件中（即一些列是特征，有一列是用于分类的标签索引）。

在示例 F-5 中，我们使用前面的（非序列）CSV 示例来处理一个分类问题，其中类别标签是在 0 和 numClasses-1（包含）之间的整数值。

示例 F-5　用 DataVec 加载序列数据

```
//首先指定包含CSV文件的根目录

File baseDir = new File("/path/to/base_directory/");

//第二步，创建并初始化序列记录读取器

//使用随机数字生成器来打乱顺序

InputSplit inputSplit = new FileSplit(baseDir, new Random());

int numLinesToSkip = 0;        //可选项，是否允许跳过标题行

String delimiter = ",";        //逗号分隔文件

SequenceRecordReader reader = new CSVSequenceRecordReader(numLinesToSkip,
delimiter);

reader.initialize(inputSplit);

//创建用于训练的DataSetIterator：

int minibatchSize = 10;        //每个小批量中的样本数

int labelIndex = 7;            //包含标签的列的索引

int numClasses = 5;            //类别数量（标签分类）

boolean regression = false;

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, numClasses,
        labelIndex, regression );

DataSetIterator iterator =
    new SequenceRecordReaderDataSetIterator( reader, minibatchSize, labelIndex,
        numClasses );

//使用DataSetIterator训练网络

myNetwork.fit(iterator);
```

SequenceRecordReader 还支持其他用途，例如使用不同的 SequenceRecordReader（如对不同的文件）进行回归，以及加载特征和标签。

F.5　数据转换：使用DataVec加工数据

通常机器学习数据的格式需要某种程度的预处理才能使用。这种预处理可以是简单地从数据集中删除一些不必要的列，也可以像从多个独立数据源连接和清理数据一样复杂。DataVec 为流水线的这个阶段的标准数据和序列数据提供广泛的转换功能。

使用 DataVec 加工数据的典型工作流如下。

(1) 为原始输入数据定义一个模式（稍后详细说明）。
(2) 定义一组在数据上执行的操作（删除列，处理缺失值等）。
(3) 加载数据。
(4) 执行操作。
(5) 保存处理后的数据。

当处理需要预处理的数据时，建议将数据预处理和网络训练分为独立的步骤。也就是说，建议先执行预处理，将数据存储到磁盘，然后在网络训练需要时从磁盘加载数据。

为了在数据集上执行一组操作，可以使用 Apache Spark，这样就能在集群和单机上执行（通过 Spark 本地模式）。此外，通过在 Spark 上执行操作，我们可以扩展到大型数据集（数亿条记录，甚至更多）。尽管 Spark 执行是当前唯一的选择，但是请记住，DataVec API 本身独立于运行平台，将来会支持其他集群计算框架。

F.5.1　DataVec转换：关键概念

对于转换功能，需要了解 DataVec 的一些关键的设计思想和类。

首先，DataVec 中的样本以与前一节相同的方式表示：每个样本都是表示标准数据的 List<Writable>，或表示序列的 List<List<Writable>>。在每个操作之后，新数据以其中一种格式来表示。这也意味着可以使用 RecordReader 和 SequenceRecordReader 类加载想转换的原始数据。

DataVec 中的 Schema 是一个定义了三种数据的类。

- 每列的名称。
- 每列的类型（数字、分类、字符串等）。
- 对每列的允许值的限制（如果有的话）（例如列是否必须仅包含正值）。

对于序列数据，我们有 SequenceSchema，它包含与普通 Schema 相同的每列的信息。DataVec 跟踪 Schema 在每个操作之后的变化，可以在执行转换操作之后或在过程中的任何点获得 Schema。

TransformProcess 定义了在数据上执行操作的次序，由指定的两种数据来构造。

- 初始输入数据的 Schema。
- 我们希望执行的一组操作。

这些是通过建造者模式来指定（如下面的示例所示）的，你也可以选择使用 JSON 或 YAML 文件。

DataVec 提供了可以在数据上执行的多种操作，见表 F-1。

表F-1：DataVec操作类型

名称	描述	示例应用
转换	每个样本或每个序列的一般操作	移除列，数学运算，解析数据 / 时间值
过滤	移除与条件匹配的样本	过滤掉包含缺失值或无效值的样本
归约	通过 key 和归约对样本分组	求每个客户 ID 的最小值、最大值或总和
转换为序列	通过一个或多个关键列将多个样本组合为序列	日志数据：将每个 IP 或客户的不同记录组合为一个序列
从序列转换	将序列数据中的每个时间步分成单独的（非序列）格式	将自定义事务的序列分为单独的记录

F.5.2 DataVec转换功能：一个示例

本节的示例非常简单，展示如何对小型数据集执行一些常见操作。

我们假定这个示例中有一些事务数据，希望从中预测标签（欺诈 / 合法），进而假定有以下简单的数据集，其中有这些列：[customerID, dateTime, amount, label]：

```
3420348,2016-01-01 06:55:07,150.00,legitimate
9087434,2016-01-01 15:16:18,78.10,legitimate
4530843,2016-01-02 11:39:24,780.83,fraud
```

首先为数据指定一个 Schema，操作如下：

```
Schema schema = new Schema.Builder()
.addColumnLong("customerID")
.addColumnString("dateTime")
.addColumnDouble("amount")
.addColumnCategorical("label", Arrays.asList("legitimate","fraud"))
.build();
```

注意，列是按照它们在文件中出现的顺序来指定的。

但是这个数据集不能直接使用。为了训练神经网络，输入数据需为数字类型数据。为了准备训练数据，需要做以下工作：

- 删除客户 ID 列；
- 将类别（字符串）标签转换为整数（0 或 1）；
- 解析 dateTime 列，并提取当天的小时作为新的特征列。

可以使用 `TransformProcess` 来指定所有这些操作，代码如下所示：

```
TransformProcess process = new TransformProcess.Builder(schema)
.removeColumns("customerID")
.categoricalToInteger("label")
.stringToTimeTransform("dateTime","YYYY-MM-dd HH:mm:ss", DateTimeZone.UTC)
.transform(new DeriveColumnsFromTimeTransform.Builder("dateTime")
```

```
        .addIntegerDerivedColumn("hourOfDay", DateTimeFieldType.hourOfDay()).build())
    .removeColumns("dateTime")
    .build();
```

每个操作将按照指定的顺序执行，这些操作中的大多数都如其名所示。对于那些含更详细参数的操作（比如 `stringToTimeTransform` 操作），请参考 DataVec 的 Javadoc。

最后必须加载数据，执行刚才定义的操作，然后保存数据。代码如下所示：

```
//对Apache Spark的一些初始化设置：
SparkConf conf = new SparkConf();
conf.setMaster("local[*]");
conf.setAppName("DataVec Example");
JavaSparkContext sc = new JavaSparkContext(conf);

//使用Spark加载数据
String path = "/path/to/my/file.csv";
JavaRDD<String> lines = sc.textFile(path);
JavaRDD<List<Writable>> examples =
    lines.map(new StringToWritablesFunction(new CSVRecordReader()));


//执行操作
SparkTransformExecutor executor = new SparkTransformExecutor();
JavaRDD<List<Writable>> processed = executor.execute(examples, process);

//保存处理后的数据：
JavaRDD<String> toSave = processed.map(new WritablesToStringFunction(","));
toSave.saveAsTextFile("/path/to/save/to/");
```

执行这些操作后，数据如下所示：

```
6,150.00,0
15,78.10,0
11,780.83,1
```

上面输出数据的列是"hourOfDay""amount"和"label"。可以使用 `process.getFinalSchema()` 从 Schema 获得输出数据的列名和类型。

这样就可以了。通过短短几行代码，我们加载了数据（以可以扩展到非常大的数据集的方式），执行了许多预处理操作，并导出了可用于训练的数据。虽然示例很简单，但展示了 DataVec 在准备机器学习数据方面的实用性和灵活性。

附录G

从源代码构建DL4J

一些开发者可能选择构建自定义的扩展、修改 DL4J 的核心，或者使用最新的代码库。这里提供给这些实践者如何直接从源代码构建 DL4J 的说明。

GitHub 是一个基于 Web 的版本控制系统，是大多数开源项目事实上的托管主机。如果你打算通过修复漏洞和提交代码来为 ND4J 或 DL4J 项目做贡献，那么你需要 Git 和 GitHub。



你真的需要从源代码开始构建吗？

如果只是打算简单地使用代码库，那么你不需要安装 GitHub，因此也无须下载源代码。

G.1　验证Git是否已安装

若要验证 Git 是否已安装并可以正常工作，请在命令行中输入以下内容：

```
git --version
```

如果命令返回错误，建议你安装 Git。如果你还没有 GitHub 账户，建议注册一个，注册免费且很简单。

G.2　克隆GitHub上核心的DL4J的工程

要想使用源代码，你需要克隆 ND4J 或 DL4J，在控制台中输入以下命令：

```
git clone https://github.com/deeplearning4j/nd4j

git clone https://github.com/deeplearning4j/datavec

git clone https://github.com/deeplearning4j/deeplearning4j
```

你可能还需要克隆 DL4J 示例代码，以便它们能够与 ND4J 或 DL4J 的预构建示例一起工作（版本会有所不同）。

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples
```



更多示例帮助

若想了解通过 Git、IntelliJ 和 Maven 安装示例代码的做法，请参阅快速入门的页面（https://deeplearning4j.org/docs/latest/deeplearning4j-quickstart）。

G.3　通过压缩文件下载源代码

另一种获取源代码的方法是点击 ND4J GitHub 页面（https://github.com/deeplearning4j/nd4j/archive/master.zip）的“Download ZIP”按钮。

G.4　使用Maven构建源代码

你可以将 Maven 和 Git 结合使用，确保 ND4J、DataVec 和 DL4J 被正确构建。为了确保你本地的这些库为最新工作版本，请切换到它们的根目录，并在提示符中输入以下命令：

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

按照 ND4J、DataVec 和 DL4J 的顺序运行一个干净的安装是获得最新的漏洞修复和特性的好方法。

附录H

设置DL4J项目

DL4J 是一套工具，为深度学习提供了一个完整的平台。你可以将多个依赖连接在一起，来执行支持深度学习模型的不同功能。DL4J 使用 Maven 来控制项目中依赖的连接。本节介绍一些相关的依赖，你可以用它们来构建自己的深度学习模型、工具和集成。

H.1 创建一个新的DL4J项目

DL4J 是一个面向那些熟悉生产部署、IntelliJ 等 IDE 和 Maven 等自动化构建工具的专业 Java 开发者的开源项目。如果你熟悉那些工具，那么我们的工具将为你提供最好的体验。我们的向量化库 ND4J 和 DataVec 将通过后面的快速启动指令自动安装。

以下是系统配置要求。

(1) Java 7 或以上。
(2) Maven 3.2.5 或以上（依赖管理和自动构建工具）。
(3) Git。

你还可以执行一些可选步骤，包括安装以下工具：

- 用于 GPU 的 Cuda 7；
- Scala 2.10.x；
- Windows；
- GitHub。

我们从 Java 开始设置环境。

H.1.1 Java

Java 是 ND4J 上主要的接口及网络开发的语言，因为它可以在包含数千个节点分布的基于云的系统和低内存的物联网设备上运行。它是个“一次编写，处处运行”的语言。

要测试你的 Java 版本（或者测试你是否已安装它），请在命令行中输入以下内容：

```
java -version
```

如果你的机器上没有安装 Java 7，请在这里下载 Java 开发工具包（JDK，http://bit.ly/2ty2o5B）。对于较新的 Mac，第一行的文件名中会包含 Mac OS X（每次更新，jdk-7u 后的版本号递增），文件信息如下所示：

Mac OS X x64 185.94 MB - jdk-7u79-macosx-x75.dmg

H.1.2 使用Maven

Maven 是 Java 项目的自动化构建工具（以及其他用途）。它会找到并自动下载 ND4J 和 DL4J 项目库的最新版本（.jar 文件）。你可以在 Maven Central 仓库（https://search.maven.org/）中找到这些项目库。Maven 让可以你轻松地安装 ND4J 和 DL4J 项目，它能很好地与集成开发环境（IDE）协同工作，如 IntelliJ。

要检查 Maven 是否安装在你的机器上，以及你拥有的是哪个版本，请在命令行中输入以下命令：

```
mvn --version
```

如果你的 Maven 不是最新版本，那么需要更新它（本文撰写时，它的版本是 3.3.x）。你可以在 https://maven.apache.org/ 找到安装 Maven 的指引。

下载包含 Maven 最新稳定版本的压缩文件。在页面下方，遵循适用于你的操作系统的指引，如“基于 Unix 的操作系统（Linux、Solaris 和 Mac OS X）”。

一个最小的项目对象模型文件

要在你自己的项目中运行 DL4J，我们强烈推荐 Java 用户使用 Apache Maven，Scala 用户使用 SBT。基本的依赖及其版本如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>MyGroupID</groupId>
    <artifactId>MyArtifactId</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <nd4j.version>0.5.0</nd4j.version>
        <dl4j.version>0.5.0</dl4j.version>
```

```
        <datavec.version>0.5.0</datavec.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.deeplearning4j</groupId>
            <artifactId>deeplearning4j-core</artifactId>
            <version>${dl4j.version}</version>
        </dependency>

        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-x86</artifactId>
            <version>${nd4j.version}</version>
        </dependency>


    </dependencies>
</project>
```

项目对象模型说明 。项目对象模型（POM）文件可能很复杂，所以我们将列出一些关键配置，帮助你更好地理解要做的事情。下面说明几个核心 DL4J POM 配置条目及其在本地 maven repo 中安装的内容。

- `deeplearning4j-core`
  `deeplearning4j-core` 包含核心 DL4J 神经网络实现。
- `nd4j-x86`
  `nd4j-x86` 是驱动 DL4J 的 ND4J 库的 CPU 版本。

H.1.3 IDE

IDE 帮助你使用我们的 API，让你只需几次点击即可构建网络。建议使用 IntelliJ 或 Eclipse，它们都能与你安装的 Java 版本协同工作，并通过 Maven 处理依赖。

使用IntelliJ快速开始创建DL4J工程

IntelliJ 的免费社区版有安装说明。

安装之后，执行以下步骤来启动和运行（Windows 用户请参阅 https://deeplearning4j.org/docs/latest/deeplearning4j-quickstart 上的演练部分）。

(1) 在命令行中输入以下内容：我们目前用的示例的版本是 0.0.4.x。

```
git clone https://github.com/deeplearning4j/dl4j-0.4-examples.git
```

(2) 在 IntelliJ 中，依次点击 File（文件）→ New（新建）→ Project from Existing Sources（来自现有源代码的工程）菜单，然后使用 Maven 创建一个新项目。
(3) 选择前面示例的根目录，这将在 IDE 中打开它们。
(4) 复制并粘贴下一节将介绍的支撑 Maven 项目的 pom.xml 文件。
(5) 等待 IntelliJ 下载所有的依赖。（右下角有水平的进度条在活动。）
(6) 从左侧文件树中选择一个示例，然后点击“run”。

H.2　设置其他Maven POM

本节将介绍如何为 ND4J 快速设置 Maven pom.xml 文件，以及这个工具套件中的其他组件。

ND4J与Maven

ND4J 后端是 DL4J 神经网络背后的线性代数运算的执行者，它随芯片的不同而变化，其中 CPU 在 86 库运行得最快，而与 GPU 与 Jcublas 库工作得最好。若要在 Maven Central 上找到 ND4J 后端，请执行以下操作。

(1) 点击最新版本下的链接版本号。
(2) 在之后出现的页面左侧复制依赖代码。
(3) 将代码粘贴到 IntelliJ 项目根目录的 pom.xml 中。

nd4j-x86 后端的依赖代码如下所示：

```
<dependency>
    <groupId>org.nd4j</groupId>
    <artifactId>nd4j-x86</artifactId>
    <version>${nd4j.version}</version>
</dependency>
```

ND4J 后端 nd4j-x86 库适用于所有示例。如果要安装额外的依赖，OpenBlas、Windows 和 Linux 用户请参考 DL4J 官网的入门介绍。

附录I

为DL4J项目设置GPU

作者：Vyacheslav Kokorin，Susan Eraly

训练神经网络涉及大量的线性代数计算。GPU 具有数千个核心，擅长解决这些问题，因此它们通常用于加速训练，并最终使你花费的每元钱、每度电都能够获得更好的计算性能。

I.1 将后端切换到GPU

DL4J 可以在 NVIDIA GPU 上工作，目前支持 Cuda 7.5，当 Cuda 8.0 发布后 DL4J 会马上支持它。我们将 DL4J 设计为即插即用，这意味着将计算层从 CPU 切换到 GPU 与切换 pom.xml 文件依赖中 `nd4j` 下的 `artifactId` 行一样简单。

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.nd4j</groupId>
            <artifactId>nd4j-cuda-7.5-platform</artifactId>
            <version>${nd4j.version}</version>
        </dependency>
    </dependencies>
</dependencyManagement>
```

I.1.1 选择GPU

一般推荐高端消费级型号或专业的 Tesla 系列的型号，在本文撰写时，两张 NVidia GeForce GTX 1070 GPU 显卡是起步阶段的可靠选择。

以下是购买 GPU 时需要考虑的重点。

- 板上集成的多处理器（或核）的数量

  板上集成的多处理器（或核）的数量越多越好，就这么简单，这涉及 GPU 在任意给定时刻能处理多少个并行线程。

- 设备上可用的显存

  设备上可用的显存涉及可以上传多少数据到设备进行处理。不过，显存类型也很重要，因为它决定显存带宽，进而决定传输速度。配置不能低于 GDDR5，GDDR5X 更好，而 HBM/HBM2 则是首选。

另一个值得一提的特性是可选的对半精度的原生支持，这一特性出现在高端 Tesla P100 设备和特殊 Tegra 设备中。根据数据维度和模型大小，该特性能够将深度学习的速度提高到 200% 到 300%。

同样重要的是考虑设备之间可用的互连选项。在本文撰写时，市场上只有两种选择，PCIe 和 NVLink，其中 NVLink 由于能够提供 160Gbps 带宽而成为明显的赢家，这使得 NVLink 成为所有可能的深度学习并行性模型在多 GPU 系统训练时的理想特性。NVIDIA 甚至提供了一个预构建服务器，内置 8 个 Tesla P100 设备，支持 NVLink 互连，被称为“DGX-1”。它的价格非常高，标榜自己很聪明，宣传词是“盒子里的超级计算机”。不管它是否真的聪明，这个宣传词很难反驳。

I.1.2　在多GPU系统上进行训练

DL4J 还支持在多 GPU 系统上以数据并行模式进行训练。DL4J 有一个 `ParallelWrapper` 类，可以用于将现有的模型转换为并行训练的模型。

例如：

```
ParallelWrapper wrapper = new ParallelWrapper.Builder(YourExistingModel)
        .prefetchBuffer(24)
        .workers(4)
        .averagingFrequency(1)
        .reportScoreAfterAveraging(true)
        .build();
```

`ParallelWrapper` 以现有模型为主要参数，并行进行训练。使用 GPU 训练时，最好将 worker 的数量设置为不少于 GPU 的数量。具体的设置值需要调整，因为它们与任务和硬件相关。

在 `ParallelWrapper` 中，初始模型将被复制，并且每个 worker 将训练自己的模型副本。在每 *X* 次迭代之后，执行 `averagingFrequency(X)`，所有模型将被平均，然后复制到 worker，训练以这种方式继续。

值得注意的是，对于数据并行训练，推荐更高的学习率。20% 左右的值会是一个很好的起始值。

I.2　不同平台上的CUDA

下面的列表列出了在不同平台上查找 CUDA 扩展指令的位置。

- Linux 上的 CUDA（http://bit.ly/2tUojGa）。
- Windows 上的 CUDA（http://bit.ly/2tUoqS6）。
- OS X 上的 CUDA（http://bit.ly/2tUojGa）。

I.3　监控GPU性能

在 GPU 上进行神经网络训练之后，你可能想了解 GPU 是否正常工作及其工作情况，下一节列出了一些帮助调试和监控 GPU 操作的资源。

NVIDIA系统管理接口

安装 NVIDIA 系统管理界面（SMI）。



使用 NVIDIA SMI

日志中会有很多信息，你可以在输出中查找“Java”，来更好地了解 ND4J 在做的事情。

附录J

解决DL4J安装上的问题

如果运行示例时出现任何错误，你需要排除一些故障。下面讨论 DL4J 的新用户会遇到的一些常见问题。

J.1 之前安装过DL4J

如果你以前安装过 DL4J，现在发现示例出现错误，请更新库。使用 Maven，只需在 pom.xml 文件中将版本修改为 Maven Central（https://search.maven.org）上的最新版本。如果使用源代码编译，你可以依次在 ND4J、Canova 和 DL4J 上运行 `git clone`，并在三个目录中运行 `mvn clean install -Dskiptests=true -Dmaven.javadoc.skip=true`。

J.2 从源代码安装时的内存错误

随着代码库的增长，从源代码安装需要更多内存。如果在 DL4J 构建过程中遇到 `Permgen error`，你可能需要增加更多的堆空间。为此，你需要找到并修改隐藏的 .bash_profile 文件，通过这个文件将环境变量添加到 bash。要查看这些变量，请在命令行中输入 **`env`**。如果要在控制台中添加更多的堆空间，请输入以下命令：

```
echo "export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=512m"" > ~/.bash_profile
```

J.3 旧版本的Maven

旧版本的 Maven，如 3.0.4，可能会出现异常，比如 `NoSuchMethodError`，可以通过升级到 Maven 的最新版本（本文撰写时为 3.3.x）来解决这个问题。要检查 Maven 版本，请在命令行中输入 **`mvn -v`**。

J.4 Maven和PATH变量

安装 Maven 后，你可能会看到这样的消息：

```
mvn 没有被识别为内部命令或外部命令、可执行程序或批处理文件。
```

这意味着你需要在 PATH 变量中设置 Maven，你可以像更改其他环境变量一样更改这个变量。

J.5 不匹配的JDK版本

如果看到以下错误：

```
配置文件 java8-and-higher 中无效的 JDK 版本：无界范围：
   [1.8, for project com.github.jai-imageio:jai-imageio-core
    com.github.jai-imageio:jai-imageio-core:jar:1.3.0
```

这说明在安装过程中你可能碰到了一个 Maven 问题，你应该将它更新到 3.3.x 版本。

J.6 C++和其他开发工具

为了编译一些 ND4J 依赖，需要安装一些用于 C 和 C++ 的开发工具。



ND4J 具体的说明

有关具体的说明，参见 ND4J 指南，网址为 http://nd4j.org/getstarted.html# devtools。

J.7 Windows和包含路径

Java CPP 的包含路径并不总是能在 Windows 上正常工作。一种解决方法是从微软 Visual Studio 的包含目录中获取头文件并将其放在已安装的 Java 运行环境（JRE）的包含目录中。



standardio.h

这将影响 standardio.h 等文件。你可以在 http://nd4j.org/getstarted.html#windows 上找到更多信息。

J.8 监控GPU

附录 I 提到过，可以使用 NVIDIA 系统管理接口（SMI）来监控 GPU。



关于监控 GPU 的更多帮助信息，请访问 http://nd4j.org/getstarted.html#gpu。

J.9　使用JVisualVM

人们使用 Java 的一个主要原因是它在 JVisualVM 中预置的诊断功能。在命令行中输入 **jvisualvm**，系统将本地 CPU、堆、PermGen、类和线程的情况可视化。



取样器视图

该工具的一个有用的视图是取样器视图。在屏幕右上方，单击取样器选项卡，然后选择 CPU 或内存按钮来显示图像。

J.10　使用Clojure

当在 Clojure 应用程序中使用 `deeplearning4j-nlp`，并通过 Leiningen 构建 uberjar[1] 时，你需要在 project.clj 中指定以下内容，以便正确地包含 akka 的 reference.conf 资源文件。

```
:uberjar-merge-with {#"\.properties$" [slurp str spit] "reference.conf"
    [slurp str spit]}
```

（注意：properties 文件映射中的第一个条目是通常的默认值）。如果没做好这一点，那么运行生成的 uberjar 时将引发如下所示的异常：

```
Exception in thread "main" com.typesafe.config.ConfigException$Missing:
    No configuration setting found for key 'akka.version'
```

J.11　OS X上的浮点支持

OS X 对浮点的支持是有问题的。当你运行我们的示例，期待数字出现，结果却看到 NAN 时，请将数据类型切换到双精度（`double`）。

J.12　Java 7的fork-join漏洞

Java 7 中的 fork-join 有一个漏洞，更新到 Java 8 可以修复它。如果出现如下所示的 `OutofMemory` 错误，说明 fork-join 存在问题。

```
java.util.concurrent.ExecutionException: java.lang.OutOfMemoryError ....java.util
    .concurrent.ForkJoinTask.getThrowableException(ForkJoinTask.java:536)
```

通过 Gitter 获得在线支持

欢迎随时在我们的 Gitter 即时聊天（https://gitter.im/deeplearning4j/deeplearning4j）中向我们咨询错误信息。

注 1：uberjar 指包含所依赖的 jar 文件的巨大 jar。——译者注

当你提交问题时，请包含以下信息（它真的会加快问题的解决！）：

- 操作系统（Windows、OS X、Linux）和版本；
- Java 版本；
- Maven 版本；
- 栈信息。

J.13　注意

下面列出了一些当 DL4J 示例不能正确构建或运行时需要检查的事项。

J.13.1　其他本地库

确保你没有在本地克隆其他代码库。主要的 DL4J 代码库正在持续改进，最新的改进可能没有完全通过示例的测试。

J.13.2　检查Maven依赖

确保示例的所有依赖都从 Maven 下载，而不是在本地查找。要删除旧的依赖，请输入以下命令：

```
rm -rf ls ~/.m2/repository/org/deeplearning4j
```

J.13.3　重新安装依赖

为了从源代码重新构建示例并确保其正确安装，在 dl4j-0.4-examples 目录中输入以下命令：

```
mvn clean install -DskipTests=true -Dmaven.javadoc.skip=true
```

J.13.4　如果出现其他错误

如果遇到问题，首先应该检查 pom.xml 文件。

J.14　不同平台

要编译某些 ND4J 依赖，你需要安装一些用于 C（语言）的开发工具，包括 gcc。要检查是否有 gcc，请在终端或命令提示符中输入 **`gcc -v`**。

J.14.1　OS X

某些版本的苹果开发工具 Xcode 会为你安装 gcc。如果你还没有 gcc，在命令提示符下输入以下命令：

```
brew install gcc
```

J.14.2 Windows

Windows 用户可能需要安装免费的 Visual Studio Community，你可以从 https://www.visualstudio.com/en-us/products/visual-studio-community-vs.aspx 下载它。



设置 PATH 环境变量

你需要手动将 Visual Studio 的路径添加到 PATH 环境变量，路径如下所示：

```
C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin
```

要通过命令提示符确认是否正确设置了 Visual Studio 的路径，请输入以下命令：

```
cl
```

你可能会收到一条消息，通知你某些 .dll 文件丢失。我们要确保 VS/IDE 文件夹在路径中（参见前面的注意事项）。如果命令提示符返回 `cl` 命令的使用信息，那就说明它在正确的位置。

1. 设置Visual Studio

要设置和配置 Visual Studio，请执行以下步骤。

(1) 设置 PATH 环境变量指向 \bin\（为了使用 cl.exe 等）。

(2) 在对 ND4J 进行 `mvn clean install` 之前，还要运行 vcvars32.bat（同样在 bin 中）以设置环境（它可以避免到处复制头文件）。



vcvars32

`vcvars32` 的作用可能是暂时的，所以需要每次在执行 ND4J 的 `mvn install` 之前运行它。

在安装 Visual Studio 和设置 PATH 变量之后，你需要运行 vcvars32.bat 来正确设置环境变量（INCLUDE、LIB、LIBPATH），这样就不需要复制头文件了。但如果你从文件浏览器运行 .bat 文件，由于设置是临时的，所以环境变量没有被正确设置。因此你需要在执行 `mvn install` 的同一个命令提示符窗口运行 vcvars32.bat，所有环境变量都将被正确设置。

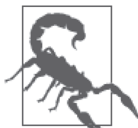
下面是设置它们的方法。

```
INCLUDE = C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\include LIB =
    "C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\lib" //这样你就可以链
                                                                 //接到.lib文件
```



记得点击 C++

使用 Visual Studio 时，你还需要点击 C++，它不再被默认设置。



Java CPP 与 Windows

此外，Java CPP（https://github.com/bytedeco/javacpp）的包含路径并不总是能在 Windows 上正常工作。一种解决方法是从 Visual Studio 的包含目录中获取头文件并将其放入已安装的 Java 运行环境（JRE）的包含目录中，这将影响 standardio.h 等文件。

2. 在64位Windows平台上运行

你可以从 http://avulanov.blogspot.cz/2014/09/howto-to-run-netlib-javabreeze-in.html 获得用于 64 位 Windows 平台的 .dll。

下载 .dll 库，并将它们放在 Java 的 bin 文件夹下（如 C:\prg\Java\jdk1.7.0_45\bin）。



其他依赖

netlib-native_system-win-x86_64.dll 库依赖：libgcc_s_seh-1.dll、libgfortran-3.dll、libquadmath-0.dll、libwinpthread-1.dll、libblas3.dll 和 liblapack3.dll（liblapack3.dll 和 libblas3.dll 只是重命名过的 libopeblas.dll 的副本）。

可以从 http://www.openblas.net/ 下载这些库。

J.14.3 Linux

如果你使用 Ubuntu 或 Centos，那么需要遵循不同的安装步骤。

1. Ubuntu

对于 Ubuntu，首先输入：

```
sudo apt-get update
```

之后，你需要运行以下命令的一个版本。

```
sudo apt-get install linux-headers-$(uname -r) build-essential
```

`$(uname -r)` 根据 Linux 版本的不同而不同。想确认你的 Linux 版本，可打开新的终端窗口，输入以下命令：

```
uname -r
```

你会看到类似于这样的输出：`3.2.0-23-generic`。无论看到什么，复制并粘贴到脚本的第一行，替换 `$(uname -r)`，然后插入一个空格并输入 **`build-essential`**，注意拼写。你可以按 tab 键来补全任何命令。

2. Centos

在你的终端（或者 ssh 会话），以 root 用户的身份输入以下命令：

```
yum groupinstall 'Development Tools'
```

之后你会看到终端上出现大量的处理和安装信息。你可以验证是否已安装了某个工具，比如对于 gcc，在终端输入以下代码：

```
gcc --version
```

访问 http://www.cyberciti.biz/faq/centos-linux-install-gcc-c-c-compiler/ 获取完整的指南。

关于作者

乔希·帕特森（Josh Patterson）目前是 Skymind 公司领域工程的负责人。Josh 曾经营过一家大数据 / 机器学习 / 深度学习领域的咨询公司。在此之前，Josh 曾是 Cloudera 公司的通用解决方案架构师，他也曾经在田纳西河流域管理局担任过机器学习与分布式系统工程师，在那里他参与的 openPDC 项目将 Hadoop 引入到智能电网。Josh 在位于查塔努加市的田纳西大学获得了计算机科学硕士学位，期间他发表了关于网状网络（tinyOS）和社会性昆虫优化算法的研究。Josh 从事软件开发超过 17 年，在开源领域非常活跃，为 DL4J、Apache Mahout、Metronome、IterativeReduce、openPDC 和 JMotif 等项目贡献过代码。

亚当·吉布森（Adam Gibson）是一位居住在旧金山的深度学习专家。Adam 曾经在财富 500 强企业、对冲基金、公关公司、创业孵化器工作，参与机器学习项目的创建。他在帮助公司处理和解析大量实时数据方面有着丰富的经验。他从 13 岁起就一直是个电脑迷，通过 http://deeplearning4j.org 为开源社区做出了积极的贡献。

关于封面

本书封面上的动物是皇带鱼（oarfish 或 Regalecus glesne），一种大型的月鱼目（条鳍鱼）鱼类，生活于温带和热带海洋。它们的身体长且细，背鳍多刺。皇带鱼可长至 11 米长，是世界上最长的硬骨鱼。

皇带鱼很少被人类看到。它们大部分时间都生活在中层海域（200 米至 1000 米水深），只有在生病或受伤时才会浮出水面。皇带鱼是食肉动物，主要以浮游动物、小鱼、水母和鱿鱼为食。

皇带鱼的肉为凝胶状，所以它不是商业渔民的目标。通常只有死去或垂死的皇带鱼被冲上岸时人类才会见到该物种。由于其大小和外形，人们认为皇带鱼可能是海蛇传说的来源。它们的总数量未知，但没有已知的对皇带鱼的环境威胁。

O'Reilly 出版的图书封面上的许多动物都濒临灭绝，它们都对世界很重要。要了解更多关于如何提供帮助的信息，请访问 animals.oreilly.com.

封面图片由 Braukhaus Lexicon 制作。



微信连接



回复“深度学习”查看相关书单



微博连接

关注@图灵教育 每日分享IT好书



QQ连接

图灵读者官方群I：218139230
图灵读者官方群II：164939616

图灵社区
iTuring.cn

在线出版，电子书，《码农》杂志，图灵访谈

O'REILLY®

深度学习基础与实践

尽管人们对机器学习的兴趣达到了一个高点，但过高的期望往往会使项目未及走远就宣告失败。机器学习，尤其是深度神经网络，如何让你的机构真正与众不同？这本实践指南不仅围绕该主题提供了实用的信息，而且帮助你开始构建高效的深度学习网络。

本书从调优、并行、向量化、构建管道等深度学习基础知识开始，逐步深入，通过现实生活中的例子，展现深度网络架构的方法和策略，并在Spark和Hadoop上使用DL4J运行深度学习工作流。

- **深入理解机器学习和深度学习基本概念。**
- **了解从神经网络到深度网络的演化历程。**
- **探索主流深度网络架构，包括卷积神经网络和循环神经网络。**
- **了解如何将特定的深度网络应用于适合的问题。**
- **全面了解通用的对神经网络和特定深度网络架构调优的基础知识。**
- **学习针对不同类型数据的向量化技术以及如何在Spark和Hadoop平台上原生地使用DL4J。**

“这正是我一直在寻找的书，它涵盖了AI开发会用到的几乎所有方法。”

——亚马逊读者评论

“这本书着重于深度学习模型的应用，并通过清晰易懂的方式来呈现。”

——亚马逊读者评论

乔希·帕特森（Josh Patterson），Skymind公司副总裁，曾任Cloudera公司通用解决方案架构师、田纳西河流域管理局机器学习与分布式系统工程师。

亚当·吉布森（Adam Gibson），Skymind公司CTO，在帮助公司处理和解析大量实时数据方面经验丰富。

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095183转600



分类建议 计算机/机器学习/深度学习

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)



ISBN 978-7-115-51542-1
定价：119.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks